

ThetaML Handbook

Stefan Dirnstorfer
Andreas J. Grau
Hongzhu Li

edition winterwork

Bibliografische Informationen der Deutschen Nationalbibliothek:

Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliographie. Detaillierte bibliographische Daten im Internet über <http://www.d-nb.de> abrufbar.

Nachdruck oder Vervielfältigung nur mit Genehmigung des Verlages gestattet. Verwendung oder Verbreitung durch unautorisierte Dritte in allen gedruckten, audiovisuellen und akustischen Medien ist untersagt. Die Textrechte verbleiben beim Autor, dessen Einverständnis zur Veröffentlichung hier vorliegt. Für Satz- und Druckfehler keine Haftung.

Impressum

S. Dirnstorfer, A. J. Grau, H. Li, ThetaML Handbook

www.thetaris.com

www.edition-winterwork.de

© 2012 edition winterwork

Alle Rechte vorbehalten.

Alle Rechte vorbehalten.

Satz: Christin Kallasch / edition winterwork

Umschlag: edition winterwork

Druck und Bindung: winterwork Borsdorf

ISBN 978-3-86468-089-2



THETARIS
Engineering in Finance

ThetaML Handbook

edition  winterwork

Stefan Dirnstorfer
Andreas J. Grau
Hongzhu Li



THETARIS
Engineering in Finance

Preface

ThetaML, the Theta Modeling Language, was born in the financial laboratories of Thetaris GmbH. Constant refinement with financial practitioners world wide has allowed us to create a financial modeling language that fits the needs of the highly agile and computationally demanding engineering tasks.

As a domain specific language, ThetaML is designed for pricing financial derivatives, selecting optimal investment strategies and managing investment risks. ThetaML employs a modular approach to separate the structure of financial products from the model stochastics and numerical details. It represents the details of financial contracts in the natural time order of contract-specified events. ThetaML programs in chronological order and streamlines code notations.

This handbook is aimed primarily at practitioners working in the financial service industry. We believe much of the material will hold significant appeal to those who are interested in exploring new approaches of financial modeling and that this book will satisfy their mental curiosity. Students and academics interested in financial engineering will find this book particularly useful for its versatile modeling approaches and the simplicity of model presentation. We hope the material will help them to approach problem solving in financial modeling in a even more flexible way.

This handbook provides a comprehensive introduction to the ThetaML language. It uses a clear and tutorial approach to explain the language syntax, enhanced with many code examples and graphical aids. It also provides tutorials that apply the ThetaML language in financial settings.

January 18, 2012

Stefan Dirnstorfer
Andreas J. Grau
Hongzhu Li



Contents

1	Introducing ThetaML	9
1.1	The ThetaML Language	10
1.2	ThetaML Language Features	12
1.2.1	Domain Specific Language for Contract Design	12
1.2.2	Programming in chronological order and computational order	14
1.2.3	Built-in conditional expectations	16
1.2.4	Implicit handling of both scenario- and time- indices	17
1.2.5	Virtual multi-threading	17
1.2.6	Pre- and post-conditions to ensure model quality	18
1.3	The Basis of ThetaML	19
1.4	The Structure of This Book	22
2	ThetaML Quick Tour	23
2.1	ThetaML Language Commands and Functions	24
2.2	Running ThetaML Example Models with Theta Suite	27
2.3	Creating and Running ThetaML Models	35
2.3.1	Creating a ThetaML Model	35
2.3.2	Running and Evaluating the ThetaML model	39
3	ThetaML Syntax Reference	45
3.1	Defining a Model	46
3.2	Using Comments	52
3.3	Assignment Operator	53
3.4	The <code>theta</code> command	54
3.5	The <code>fork ... end</code> Statement	58
3.6	The <code>if ... else ... end</code> Statement	63
3.7	Array in ThetaML	66
3.8	The <code>loop ... end</code> Statement	69
3.8.1	Fixed Length Loop	69
3.8.2	Infinite Length Loop <code>loop inf ... end</code>	71
3.8.3	Array Looping	73



3.8.4	Multiple Array Looping	74
3.9	Calling A Sub-model	76
	Implicit <code>fork ... end</code>	77
3.10	Matlab Native Access	82
3.10.1	Calling Matlab Functions	83
3.10.2	Calling a Complex Stepping Object	85
3.11	ThetaML Operators	102
	The Future Operator “!”	102
3.12	Functions	105
3.12.1	The Function <code>E()</code>	105
3.12.2	The Function <code>Beta()</code>	108
3.12.3	Other Functions	109
3.13	System Parameters	110
3.13.1	The Parameter <code>@dt</code>	110
3.13.2	The Parameter <code>@time</code>	114
3.14	Chapter Example	115
4	The ThetaML Type System	125
4.1	The Boolean Type	126
4.2	The File Type	127
4.3	The Enum Type	128
4.4	Array types	129
5	ThetaML Interfaces	131
5.1	Interface Syntax	132
5.2	Interface Import and Export Statements	136
5.3	Language Constraints	137
5.4	Value Assertions	139
6	Workflows	141
6.1	Workflow Definitions	142
6.2	Workflow Statements	144
6.3	Assignments in Workflows	145
6.4	Loops in Workflows	146



6.5	Conditional Executions in Workflows	147
6.6	Sub Workflows	148
6.7	Functions	149
6.8	External Namespaces	151
7	ThetaML Language by Example	153
7.1	Tutorial From European to American	154
7.1.1	The Stochastic Process	154
7.1.2	European Option	155
7.1.3	Bermudean Option	157
7.1.4	American Option	159
7.1.5	Compound Option	161
7.1.6	Hedged American Option	164
7.2	Tutorial Hedging in ThetaML	166
7.2.1	Introduction	166
7.2.2	Delta Hedging	167
7.2.3	Variance Minimization by Hedging	176
7.2.3.1	Single Dimensional Stochastics	177
7.2.3.2	Multi-Dimensional Stochastics	184
7.2.4	Static and Dynamic Hedging	185
7.2.5	Transaction Costs	193
8	ThetaML Tips and Tricks	201
8.1	Nested if Improves Speed	202
8.2	Reducing Variance by Hedging Improves Convergence	203
8.3	Adding Control Variables Can Improve Accuracy	205
8.5	Avoiding Direct Assignment of Expected Values Improves Accuracy	207
8.6	Keep Export Variables Unique	209
	References	211
	Index	213



Text Conventions Used in This Book:

- **Bold** text denotes the first appearance of a ThetaML language keyword or a ThetaSuite component name.
- *Italic* text is used for numbered ThetaML code examples.
- ThetaML code statements are written in a typewriter font `Courier New`.
- Colored texts are ThetaML specific commands.



1 Introducing ThetaML

This chapter familiarizes you with **ThetaML**, the Theta Modeling Language. It provides an introduction to ThetaML and its modeling features. It also provides some guidelines on the structure of this book.

This chapter includes the following sections:

- The ThetaML Language
- ThetaML Language Features
- The Basis of ThetaML
- The Structure of this Book



1.1 The ThetaML Language

ThetaML is designed to be as simple as possible while maintaining the ability to model any financially relevant task, especially in the area of using stochastic models to price financial derivatives.

ThetaML describes financial models in two forms, either text-based as **ThetaML Script** or graphically as **Thetagram**. Figure 1 illustrates the relationship among ThetaML, ThetaML Script and Thetagram.

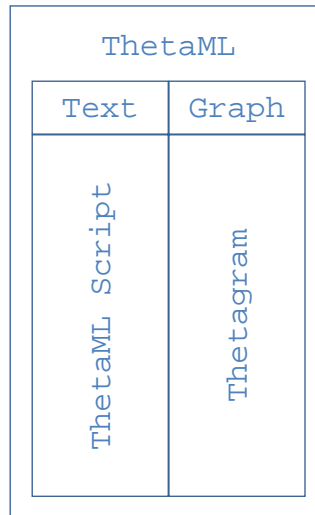


Figure 1. The Relationship among ThetaML, ThetaML Script and Thetagram: ThetaML is the XML representation of ThetaML Script and Thetagram.

Both ThetaML Script and Thetagram can capture the full details of a financial model and may be transformed into each other, allowing the user to choose a way of modeling based on personal preferences and experiences.



ThetaML Script as a compact script language allows for rapid financial model development in text-based form. Thetagram¹ as a graphical representation requires almost no programming skills and quickly yields visual presentations of financial models.

In the following, we will write ThetaML when referring to ThetaML Script or Thetagram; the exact meaning will be clear from context.

¹ In the Theta Suite, a Thetagram can be automatically created from a ThetaML model using these steps: Thetagram is created by selecting a .thetas file, right click and choose the option 'Initialize Thetagram'. This gives you a list of ThetaML model names in rectangular-shaped boxes. It includes all the models in the selected .thetas file. By double clicking one of the boxed model name, we get a display of the graphs generated for that particular model.



1.2 ThetaML Language Features

As domain specific programming language, ThetaML has both functional and procedural features. The main language features include:

- Domain specific language for contract design
- Programming in chronological order and computational order
- Built-in conditional expectations
- Implicit handling of scenario- and time- indices
- Virtual multi-threading
- Pre- and post-conditions on models to ensure model correctness

We elaborate on each feature in the following.

1.2.1 Domain Specific Language for Contract Design

ThetaML is the domain specific language which we use to describe and model the structural features of stochastic models in financial engineering.

It is designed for pricing financial derivatives, selecting optimal investment strategies and managing investment risks.

Defining and reading financial products in ThetaML is done with unprecedented simplicity and is significantly faster than using conventional term sheets.



Complex term sheets are currently the only way to express and communicate the contents of a financial product. The process of turning a term sheet into a computable pricing algorithm is long and error prone. So far, there exists no adequate standard for specifying the structural model of an arbitrary financial derivative. To address this issue, we introduce the definition language ThetaML that allows the specification of structural models in a way that is both precise and intuitive. Furthermore, details of financial contracts can be represented in the natural time order of the events.

ThetaML can be automatically translated into numerical algorithms. Hence, it is not only a description language, but also solves a specific numerical problem. Being completely independent of the numerical schemes, Monte Carlo, PDE solvers or trees could encompassingly be used to compute the desired result. A financial engineer using ThetaML to design or analyze a certain financial product can thus focus on the problem domain without having to worry about the numerical details.



1.2.2 Programming in chronological order and computational order

Programming in *chronological* event order is what distinguishes ThetaML most from other programming languages.

Figure 2 illustrates the difference between programming in chronological order and computational order, using the pricing process for an American-type option as an example. The ThetaML pseudo code for pricing the American-type option simply follows the time order in which the events of a pricing model are (expected) to occur. In contrast, code written in computational order requires significant reordering.

The unique feature of ThetaML as a programming language is the ability to access the future values of stochastic variables. This is achieved with the future operator “!”. The future operator “!” and the **theta**² command allow ThetaML programs to be written in chronological order but be computationally evaluated backward in time.

² For definitions of the future operator “!” and the theta command, please reference section 2.1.



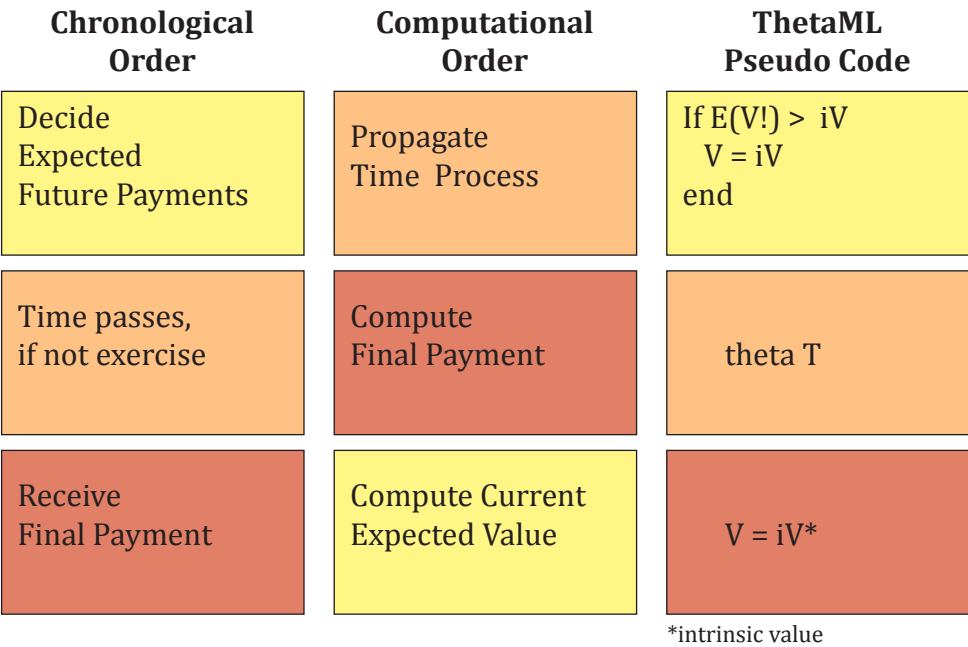


Figure 2. Illustration of programming in chronological order and computational order, with a ThetaML pseudo code for an American-Type Option.

In the graph, each text box has an event happening. Text backgrounds indicate time order: The lightest background means the events are evaluated at current time; intermediately shaded background indicates intermediate time propagation process, and deep colored events happen at final time.

In comparison, conventional programming languages can only access a value that is previously assigned to a variable and the program codes are evaluated sequentially.

ThetaML as a programming language includes the ability to program conventionally in computational order; on top of that, it adds the unique style to program in chronological order. This is facilitated with the future operator “!” and the **theta** command.

The ability to program in chronological event order is unprecedented. The future operator “!” is handy, especially in cases of pricing more complex financial derivatives with early exercise decisions, such as American options, chooser options, convertible bonds and others, when the decision-making process involves expected future values unknown at the decision-making time.

1.2.3 Built-in conditional expectations

ThetaML can evaluate the conditional properties of stochastic variables or processes as a statistical approximation using a single function - $E(\cdot)$.

In ThetaML, time is virtual and is called model time. Model time is defined in terms of chronological event-triggered order, that is, model time follows the time sequence in which chronological events occur in a pricing model. The function $E(\cdot)$ is computed conditional on all function parameters values that are known at the corresponding model time.

The arguments of the function $E(\cdot)$ implicitly access their estimated future values at future time points. The access to future values at current time is realized with the future operator “!”. Advancing model time between current and future time is done with the `theta` command.

The function $E(\cdot)$ is applicable to all process variables in ThetaML. In order to compute the expectation function $E(\cdot)$, the compiler automatically identifies the independent variables, conditional on the information known at the current model time. More precisely, the compiler determines the smallest sigma field of the underlying state variables at current model time, by filtering out the set of state variables that are measurable at current model time. Thus, the compiler hides the hassle associated with the numerical evaluation of $E(\cdot)$ and conditions it on the information known at the corresponding model time. The function $E(\cdot)$ can



be applied, for example, to the conditional evaluation process in the Least Squares Monte Carlo (LSMC) method for early exercisable options. It can also be used in the risk-neutral pricing of European-type options, in which case the function $\mathbb{E}(\cdot)$ summarizes the averaging process as what is done with Monte Carlo pricing derivatives in other programming languages.

1.2.4 Implicit handling of both scenario- and time- indices

Variables defined in ThetaML represent a stochastic process. They do not require explicit scenario- or time- indices. The scenario- and time- indices are implicitly handled by ThetaML. By writing ThetaML in chronological order, the semantic information implicit in a process variable is revealed by the model time explicitly defined in a pricing application. Consequently, the pricing task is independent of the numerical procedures used to simulate the stochastic processes.

In Monte Carlo evaluations, we often need to consider both scenario and time indices. This is where index confusions and errors might easily occur. By simultaneously taking care of both scenario and time indices, ThetaML saves users loads of efforts in cases like writing loops and evaluating conditional multi-indexed statements, besides having the desirable effect of producing compact programming codes.

1.2.5 Virtual multi-threading

ThetaML allows simulations with multiple threads (i.e. multiple simulation tasks) to be executed in parallel at model time. This is enabled with the `fork ... end` statement and the `theta` command.



As an investment portfolio typically consists of hundreds of financial products, each financial product may have some event-related peculiarities, the separation and synchronization of simulation tasks is both convenient and necessary.

In ThetaML, the `fork ... end` statement is used to couple multiple simulation tasks that occur parallel in model time; the `theta` command that defines and passes the model time is used to synchronize the simulation tasks virtually paralleled by the `fork ... end` statement.

1.2.6 Pre- and post-conditions to ensure model quality

ThetaML has interfaces to ensure that certain parameter or process values are within the range of the constraints. The constraints for initial parameter values are imposed before a model is run. If a ThetaML model does not comply with the constraints imposed by the interface, the model can not be executed and returns the violated constraint instead. The constraints on the exported process variables are imposed after a model is run. This post conditions must be satisfied by the exported variables at all model time steps.

Interfaces in ThetaML make easy the task of model reviews, such that the modeler does not have to check each `sub_model` for violations of value constraints.



1.3 The Basis of ThetaML

The essence of ThetaML is the virtual timing model it operates in.

Unlike conventional programming languages where the computational flow is determined by the order of code statements, ThetaML operates on a chronological model time axis: ThetaML allows the programmer to insert time delays between code statements at different model times using the `theta` command. The values of variables at a given line of code are evaluated at the model time associated with that line of code.

Different blocks of codes executed simultaneously have a common model time axis. The model time grid is a collection of all the event times occurred in multiple simulation tasks; different simulation tasks may have different events occurring at different time. Within multiple simulation tasks, variables can be cross-accessed and all variable values at that line of code are evaluated at the model time executed at that line of code.



The ThetaML code and Figure 3 below give an illustration for how the virtual timing model operates in ThetaML:

```

    % the fork ... end block that performs simulation task 1
1: fork
2:     theta 1.5 % 1.5 time units pass from time 0
3:     x = 1     % at time 1.5, x is assigned the value 1
4:     theta 4.5 % another 4.5 time units pass
5:     x = 4     % at time 6, x is assigned the value 4
6: end

7: % the fork ... end block that performs simulation task 2
8: fork
9:     y = 5     % at time 0, y is assigned the value 5
10:    theta 3    % 3 time units pass from time 0
11:    y = 10    % at time 3, y is assigned the value 10
12: end

13: % code statements that perform simulation task 3
14: theta 1.5    % 1.5 time units pass from time 0
15: z = x!      % at time 1.5, z accesses the future value of x

```

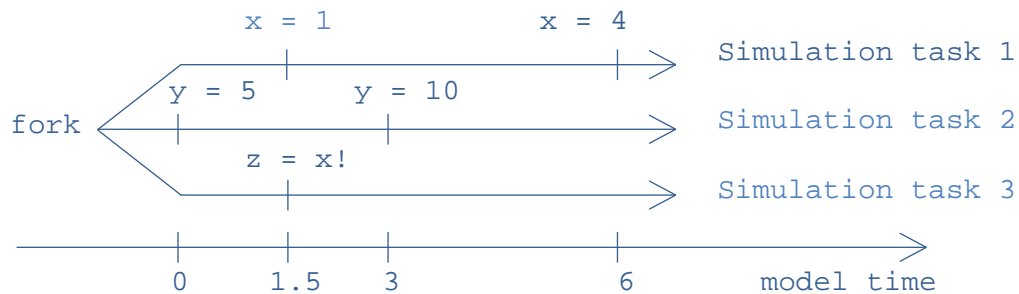


Figure 3. The virtual timing model in ThetaML.

This figure is a graphical illustration of the above ThetaML code statements.



In Figure 3, the time axis records all the model times occurred in the three simulation tasks. For example, `Simulation task 1` has an event happening at time 1.5, i.e. `x` is assigned a value of 1 (line 3 in the ThetaML code). `Simulation task 2` has an event happening at time 3, i.e. `y = 10` (line 11 in the ThetaML code). `Simulation task 3` cross assesses the value of `x` in `Simulation task 1` via `z = x!` (line 15 in the ThetaML code), i.e. the variable `z` is assigned the future value of `x` (`x = 4` at time 6) using the future operator “!”.

In ThetaML, the simulation tasks are virtually paralleled using the `fork ... end` statement and synchronized by the `theta` command. As a result, the model time axis combines all the event time occurred in the three simulation tasks and it is shared by the three simulation tasks. Time passing along the model time axis is enabled with the ThetaML command `theta`.

This virtual timing model of ThetaML is the key to its ease of use in describing financial derivatives. Since financial derivative contracts typically have sequential time-triggered events, such as scheduled payments, a maturity time, etc., ThetaML can simulate this type of multiple-event sequential-time processes. The ability to execute multiple code elements in parallel allows the users to model cross-dependent financial products or variables. An option on a bond can be simulated in a way such that both processes - the option and its underlying bond - evolve as what they would evolve in real-time financial markets.



1.4 The Structure of This Book

The plan of this book is as follows. After introducing ThetaML in Chapter 1, we provide in Chapter 2 a quick tour of ThetaML. Chapter 3 documents the detailed ThetaML language syntax and provides many code examples. The Chapter Example gives a first application of ThetaML in financial modeling. Following that, in Chapter 4 we detail on the ThetaML type system. Chapter 5 documents ThetaML interfaces. Chapter 6 writes about ThetaML workflows. Chapter 7 provides in ThetaML two tutorial examples in the area of pricing and hedging financial derivatives. Finally, in Chapter 8 we give some tips and tricks for more efficient use of ThetaML in financial applications.



2 ThetaML Quick Tour

This Chapter gives an overview of

- ThetaML Language commands
- Running a ThetaML example in Theta Suite
- Creating the first ThetaML model and evaluating it with Theta Suite



2.1 ThetaML Language Commands and Functions

The following is a selected list of ThetaML language concepts, commands and functions.

- **model time**
In a ThetaML model, time is virtual and is called model time. Model time is defined in terms of chronological event-triggered order. Model time proceeds forward in time according to the time sequence of events that occur in a pricing model. Model time is passed by the **theta** command.
- **the **theta** / **Theta** command**
The crucial **theta** command defines and passes model time. Model time is used to synchronize multiple simulation tasks that occur parallel in model time. Every **theta** command is followed by a statement defining a time interval.
- **the **fork ... end** statement**
With the **fork ... end** statement, code blocks are virtually executed in parallel in model time.
- **the future operator “!”**
The unique future operator “!” allows access to the future values of a variable.
- **the infinite loop **loop inf ...end****
The flexible **loop inf** allows the loop to run until all other fixed-length loops sharing the same model time axis are terminated by their model determined length.



- the fixed loop `loop ... end`
The `loop ... end` statement allows repeated executions for a fixed number of times. The parameter after the `loop` statement defines the type of loops. It can be an integer for a finite number of iterations, or an array, in which case the loop will iterate over the elements of the array.
- the function `E()`
The `E()` function computes the conditional expected value of a variable or an expression.
- the function `Beta()`
The `Beta()` function takes two arguments and computes the beta factor(s) between these two arguments, conditional on the current information.
- the parameter `@dt`
The time interval parameter `@dt` has different values depending on its context. If `@dt` follows the `theta` command, it evaluates to the time that passes to the next event-triggered time. In case `@dt` is found elsewhere, it evaluates to the time step that has passed since the thread's previous `theta` command. The `@dt` parameter is most often found within the infinite loop `loop inf ... end`.



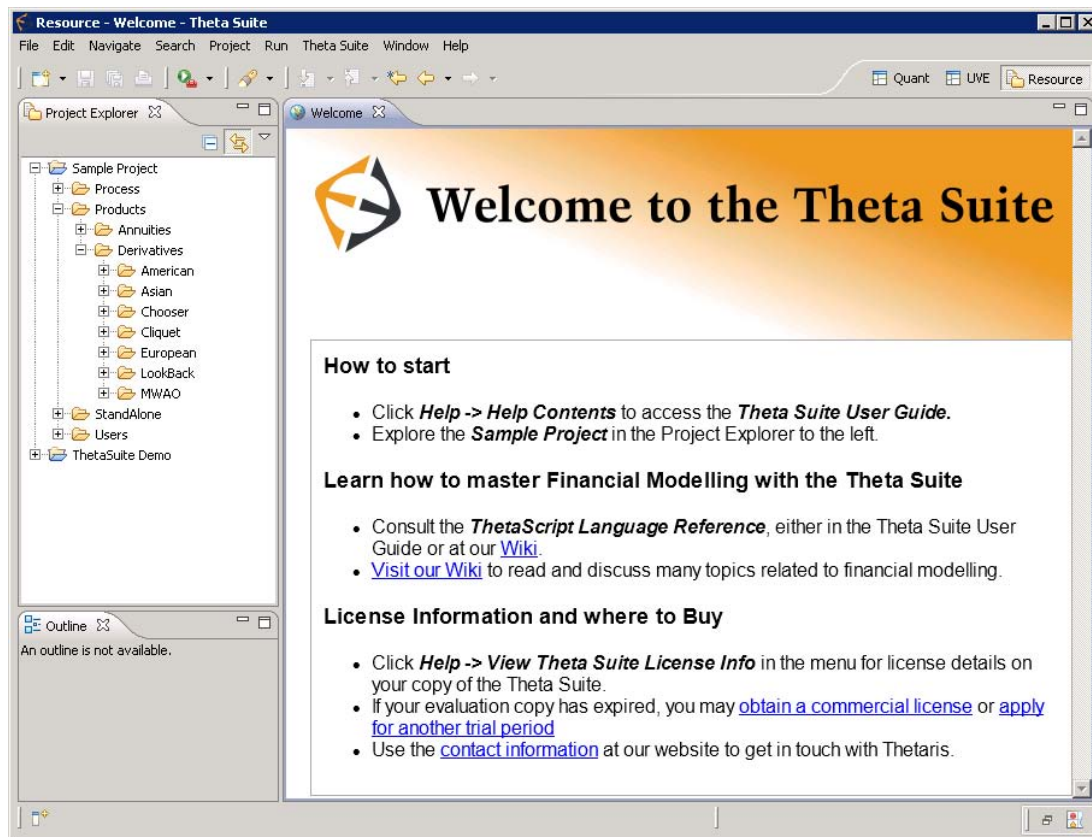
- the parameter **@time**
The **@time** parameter variable returns the current model time. It is the sum of all previous **theta @dt** time steps. The **@time** parameter is most often found within the infinite loop **loop inf ...end**.
- the parameter **@scenarioIndex**
The parameter **@scenarioIndex** extracts the index of current Monte-Carlo scenario.
- the parameter **@scenarioSize**
The parameter **@scenarioSize** extracts the size of Monte-Carlo scenarios.



2.2 Running ThetaML Example Models with Theta Suite

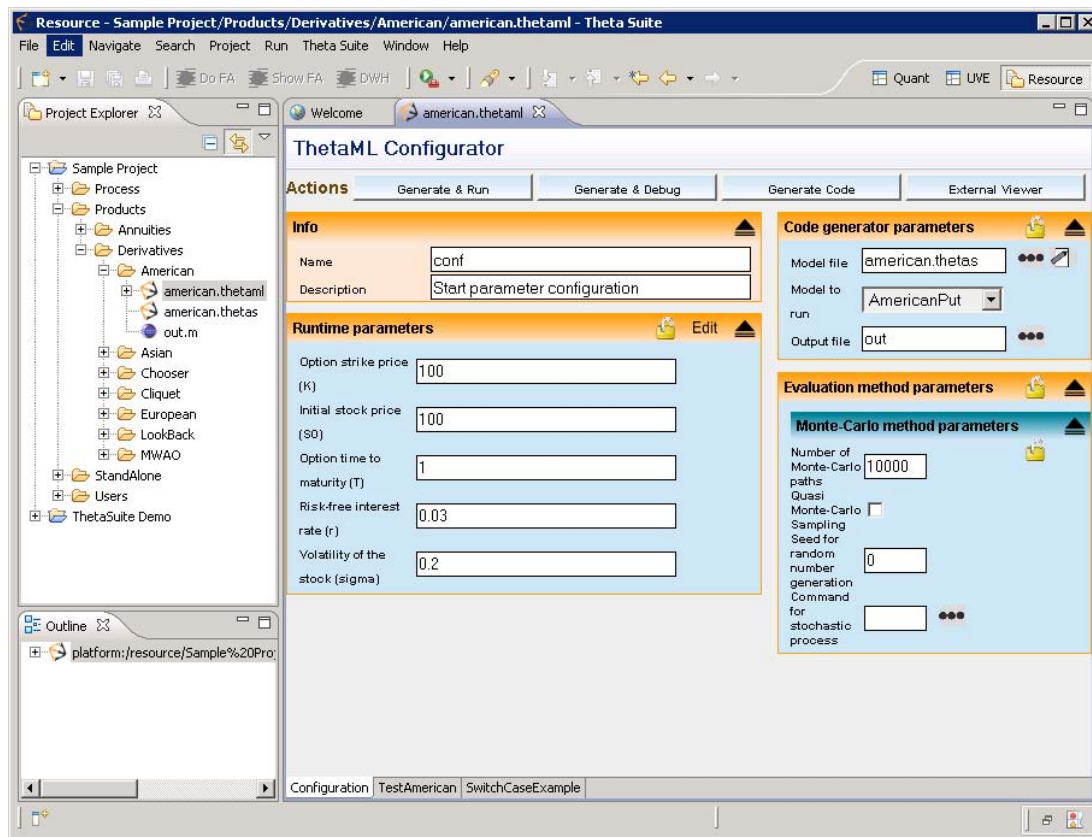
This section gives an introduction to running example models in Theta Suite.

- 1) We can start run “Theta Suite” from the “Start” menu or by double clicking “ThetaSuite.exe”. After a few seconds’ loading, we have in front of us the Welcome page of Theta Suite.



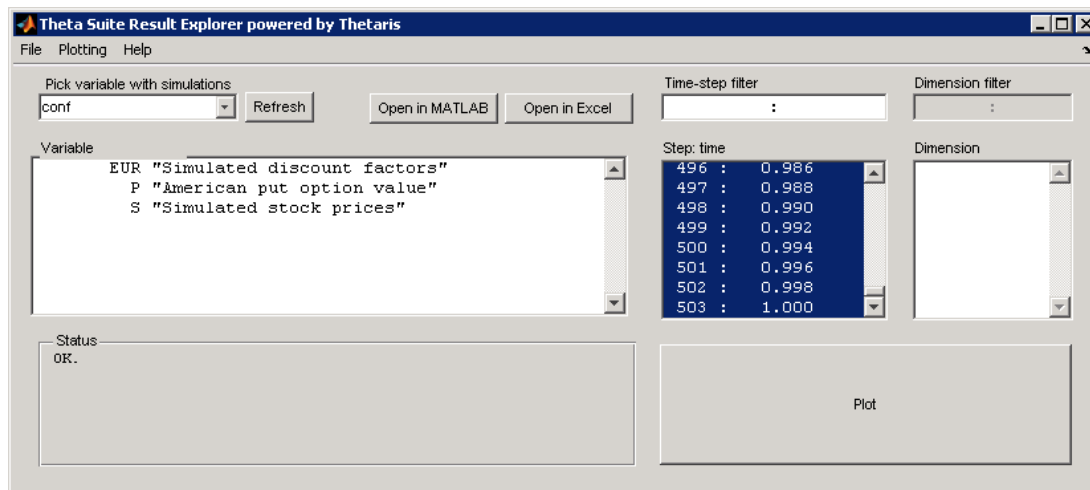
- 2) In the “Welcome” view of Theta Suite, there are sections that guide you to find information about how to start using Theta Suite and how to write ThetaML models.
- 3) Every installed ThetaSuite.exe has a “Sample Project” folder that provides the user a list of ThetaML model examples. We can find the “Project Explorer” view on the upper left corner of the Theta Suite workspace. In case the “Project Explorer” view is not already open, we can open the view from the menu “Window”, then choose “Show View”, and select “Project Explorer”.
- 4) In the “Project Explorer” view, we can open the folder “Sample Project”, then choose the subfolder “Products”. Under the subfolder “Products”, choose the sub_subfolder “Derivatives”, there is a list of folders named after various option types, such as “European”, “American”, “Asian”, “Chooser”, and so on. Open the folder “American”, by clicking on the boxed “+” sign. This expands into a list of files, including the ThetaML configuration file ended with type .thetaml, the ThetaML model file ended with type .thetas, or Thetagram file ended with type .thetagram.
- 5) Select and double click the runtime configuration “american.thetaml”. This brings into view the ThetaML Configurator page:





- 6) To start model evaluation, we give some values for the input parameters in the field “Runtime parameters”. We then specify the number of simulation paths and provide a random seed for the “Monte-Carlo method parameters” under the field “Evaluation method parameters”. In the field “Code generator parameters”, we can browse the existing model files and select the one we wish to evaluate – american.thetas - for “Model file”. This automatically brings up a drop box list of model names for “Model to run”, choose the model named “AmericanPut”. The default name for the “Output file” is “out” with type .m, i.e. the generated output file is a Matlab m-file.

- 7) To run the model “AmericanPut”, click the button “Generate & Run” in the “Actions” bar. This leads to a status window titled “Running generated code...”, the progress bar in the status window indicates the progress of the running process. The compiler first optimizes the code execution order, then compiles and runs the codes. After the code optimization, compiling and running process, the “Theta Suite Result Explorer” comes up and shows a list of named variables exported by the “AmericanPut” model in the file “american.thetas”.



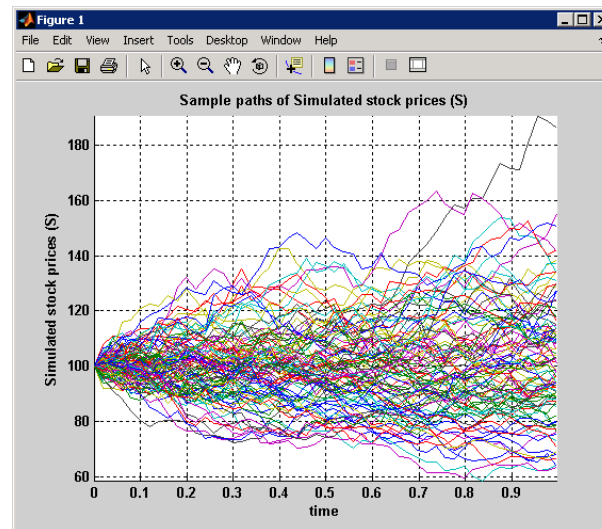
- 8) The “Theta Suite Result Explorer” is a versatile tool with many options for analyzing simulation results. Simply select a variable, the “Status” field shows a summary of the simulated results for the selected variable. This includes the number of simulation paths, the dimension of the variable, and the mean value of the variable plus and minus two standard errors.

Select the variable “S”, we have in the field “Step : time” a list of ‘index - model time’ pairs: the left-hand side of “:” marks the index for “S” at a certain model time, the right-hand side of “:” are the model times of “S”. If we select the index

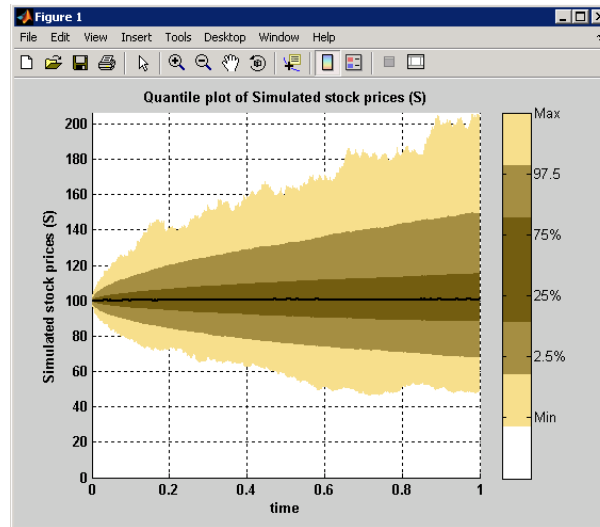


step “10” at model time “0.018”, the “Mean” value under the “Status” field automatically changes to the mean value of “S” plus and minus two standard errors at this model time, this average is taken across all the Monte-Carlo paths. We can also press the “Plot” button and bring up the “Histogram” plot of the stock “S” distributed across all Monte-Carlo paths at model time “0.018”.

If we are interested to know the values of “S” at a selected number of time steps, such as the values of “S” at every ten model time steps, we can filter the values of “S” in the field “Time-step filter” by using the range specifier: [from : step_size : to], in this case it is 1 : 10 : 503, then press enter. As expected, the “Time-step filter” selects every 10-th of the “index - model time” pairs. Simply press the “Plot” button, we have the following screen shot of the “Sample paths of Simulated stock prices (S)”, for values of “S” spaced at every 10-th model time steps:



To see the quantile distribution of Stock “S”, select “S” in the “Theta Suite Result Explorer”, right click and choose “Plot” then “Quantiles”, this brings up the quantile plot of stock “S”:



If the Stock “S” is two dimensional, we can use the “Dimension filter” field to single out stock values for either the first or the second dimension, by using the range specifier 1:1 or 2:2 respectively.

- 9) The “Theta Suite Result Explorer” provides two other options for further examinations of the simulation results. We can either “Open in Matlab” or “Open in Excel” the output data, by pressing the respective buttons.

The “Open in Matlab” button brings up the Matlab console window. Select the “Workspace” tab, we have the “conf” struct in view. Double click on the “conf” name and bring up the “Array Editor” for “conf”. The “Array Editor – conf” has the same fields as the ThetaML Configurator. The “conf” fields are stored as Matlab struct type. Double clicking any of the field values shows the subfields of yet another struct type, simply double clicking again any of the field values, we eventually arrive at the parameter values initially inputted by us in the ThetaML Configurator.



The “Open in Excel” button leads us to an export dialogue box, which asks for the export type for the output data. Select for example “Excel plain”, we have the input fields respectively for “File name” and “Scenario range”. We can either give a new name for the output file or browse and select an existing file to replace the old data. The default “Scenario range” for the simulation outputs is one scenario, we can selectively specify the “Scenario range” that we wish to export by using the range operator [from : step_size : to], such as [1 : 2 : 10], that is, every second scenario of the first 10 scenarios. Pressing either the button “Export to file” or the button “Open in Excel” exports the output data to the desired destination. The “Export to file” option shows the exported file name in the current directory, i.e. in the folder “American”; the “Open in Excel” option directly opens the Excel workbook with one worksheet named “Mean” and a number of other worksheets named “Scenario_#”, where the # sign represents Scenario number. For example, the worksheet name “Scenario_1” refers to the first scenario in the simulation output, the worksheet name “Scenario_3” refers to the third scenario in the simulation output, and so on. The worksheet “Mean” gives the mean values for the exported variables at each model time, the mean is taken over all the Monte-Carlo paths. The worksheet “Scenario_1”, for example, has the simulated values for the exported variables at each model time in the first scenario.



- 10) The “Theta Suite Result Explorer” shows in the “Pick variable with simulation” field a list of the saved configuration variables, selecting any variable from the list brings up the saved simulation output for variables exported by that configuration variable. The option to switch easily from one saved output to another helps to compare simulation results without having to generate and run the codes each time.
- 11) We can alternatively export and save output data using the menu “File”, and plot the graphs using the options provided by the menu “Plotting”.

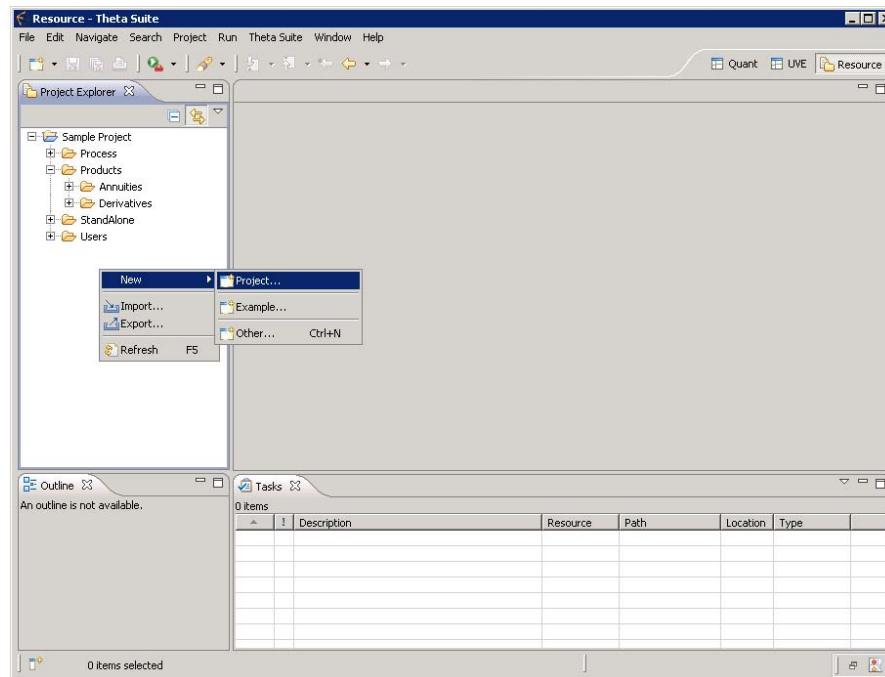


2.3 Creating and Running ThetaML Models

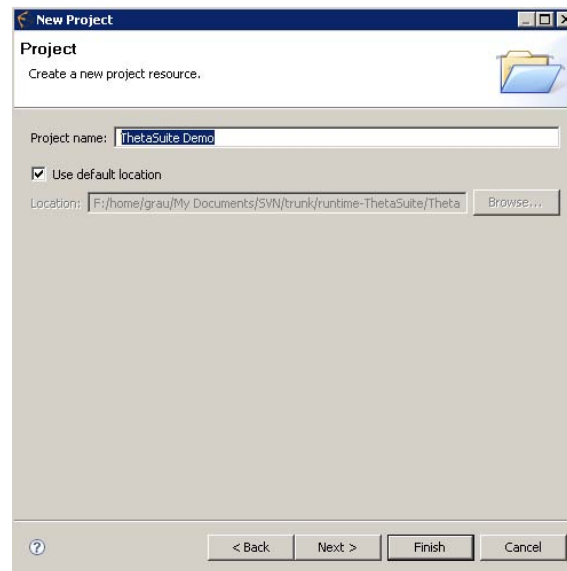
This section shows you how to create and run your own ThetaML model in Theta Suite.

2.3.1 Creating a ThetaML Model

To start with, we create a project to hold the new file. Right click a space in the “Project Explorer” view, select “New” then “Project...”:



This brings out the “New Project” wizard window. Select “General” then “Project”, and press the “Next” button. In the next “New Project” window, give a name for the new project, such as “ThetaSuite Demo” in the following picture:



Then press the “Finish” button, the created new project shows up in the “Project Explorer” view along with other existing projects.

We can now create a model file within the new project. Select and right click the “ThetaSuite Demo” project, select “New” then “File”.

In the “New File” dialog window, select the parent folder “ThetaSuite Demo”, type in the file name “demo.thetas”:





After pressing the “Finish” button, we have an empty editor.



We can now start writing ThetaML models in the editor. The following is an example ThetaML model for simulating stock prices “S” that are driven by a Geometric Brownian motion process:

```
% This model simulates a stock price process 'S'.
% The stock prices 'S' follow a Geometric Brownian motion
% process. The model name is 'GBM', it imports a
% parameter named 'S0' as the initial stock price,
% and exports in the variable 'S' the simulated
% stock price process
model GBM
    import S0 "Initial stock price"
    export S "Simulated stock price process"

    % setting the stock price parameters
    % 'mu' is the drift of the stock prices 'S'
    mu = 0.05

    % 'sigma' is the volatility of the stock prices 'S'
    sigma = 0.4

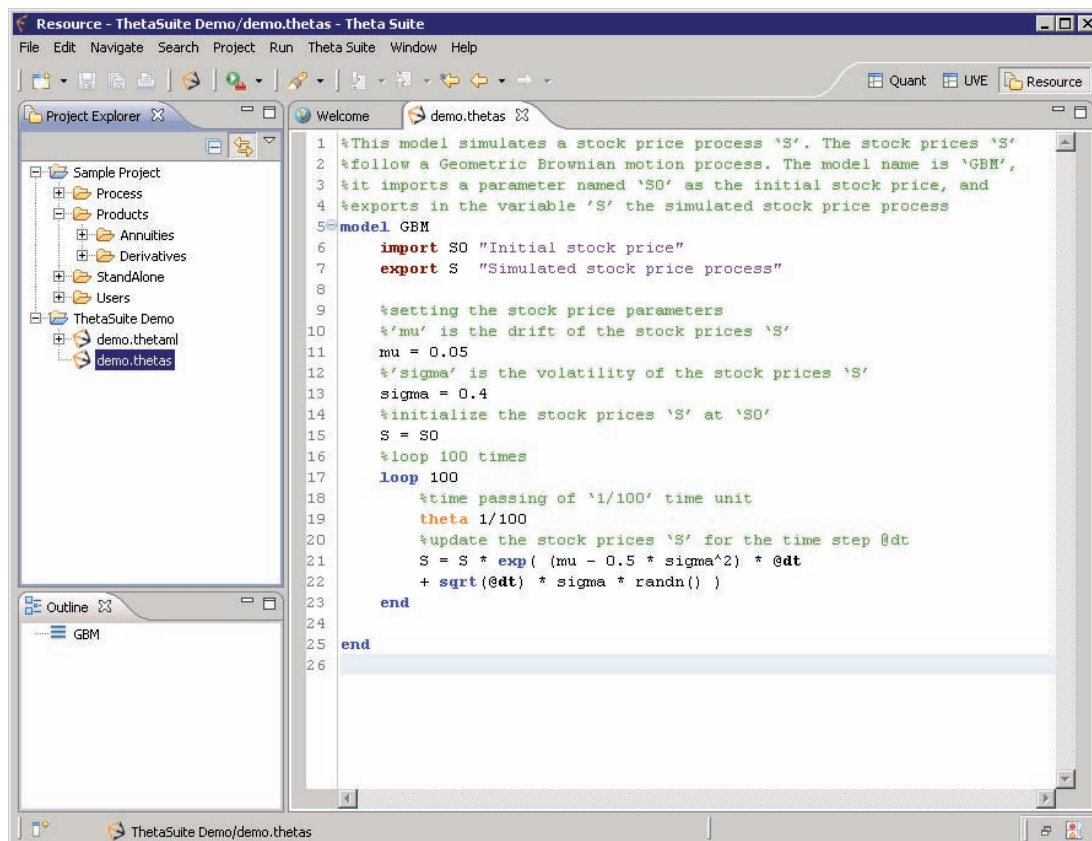
    % initialize the stock prices 'S' at 'S0'
    S = S0

    % loop 100 times
    loop 100
        % time passing of '1/100' time unit
        theta 1/100


        % update the stock prices 'S' for the time step @dt
        S = S * exp( (mu - 0.5 * sigma^2) * @dt
            + sqrt(@dt) * sigma * randn() )
    end
end
```

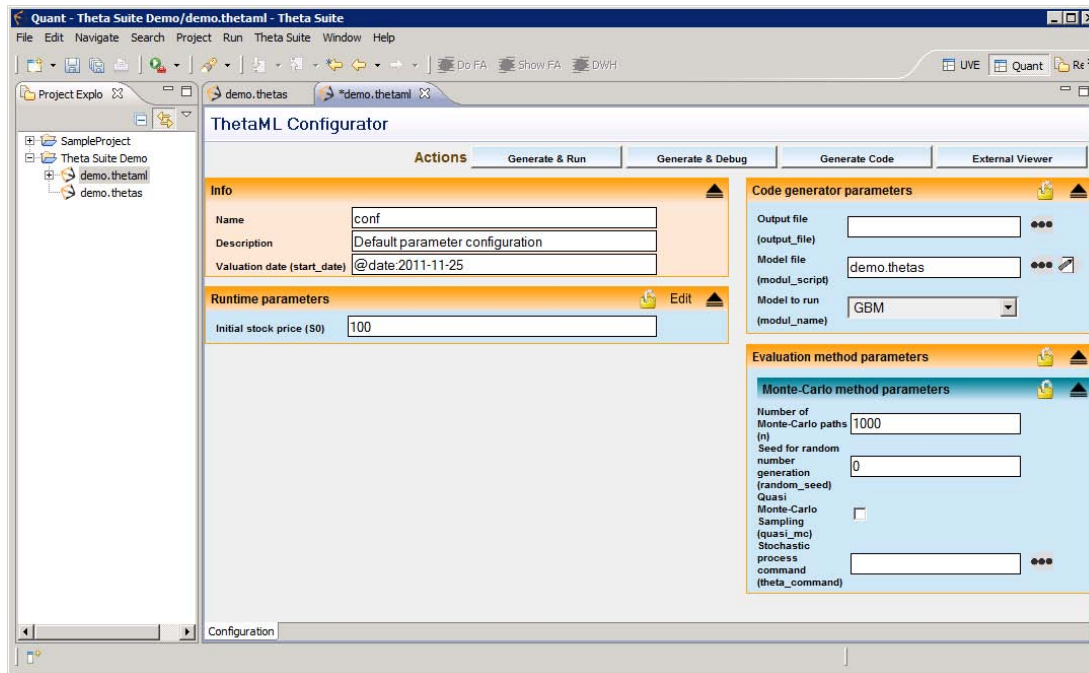
Copy and paste the above ThetaML model GBM into the editor, we have the following screen shot:





2.3.2 Running and Evaluating the ThetaML model

To evaluate the ThetaML model GBM, save the model file "demo.thetas". Then create a configuration file called "demo.thetaml". The configuration file is automatically created by clicking the "ThetaML" button  on the menu bar in the above screen shot. The configuration file can also be created by selecting the "demo.thetas" model file in the "Project Explorer" view, right clicking and choosing the option "Initialize ThetaML Configuration". Either way, we have the following "ThetaML Configurator" page:

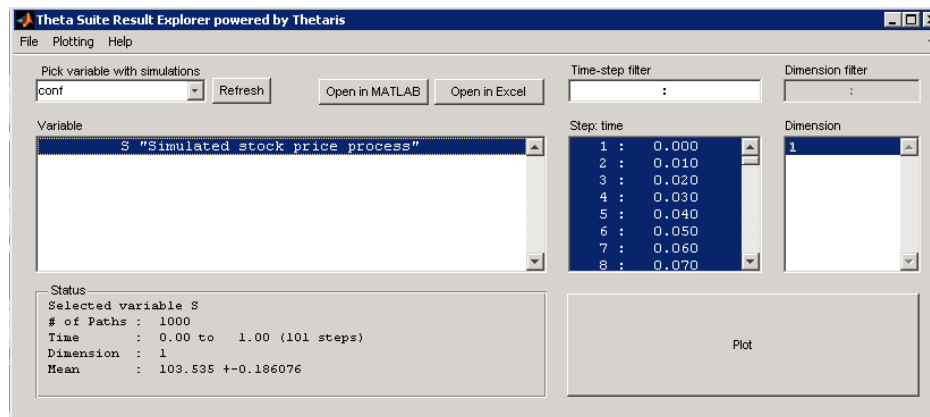


In the configuration editor of “demo.thetaml”, we give a value for the “Initial stock price (\$0)” in the field “Runtime parameters”, specify the “Number of Monte-Carlo paths” and set the “Seed for random number generation” in the field “Monte-Carlo method parameters”. Then select or type in “demo.thetas” for the “Model file”, choose the ThetaML model “GBM” for the “Model to run”.

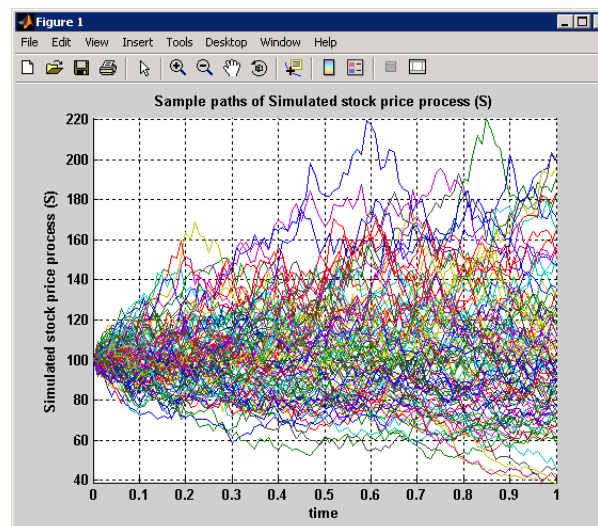
To start the model evaluation, click the button “Generate & Run” in the “Actions” bar:

After the code compiling and running process, the “Theta Suite Result Explorer” shows up. The screen shot looks like this:





We can examine the simulation results for stock prices “S”. Select the variable “S”, right click and choose “Plot”, then “Sample paths”, this produces a sample path plot of stock prices “S” as shown in the following figure:



This figure can alternatively be produced by selecting the variable “S” then pressing the button “Plot”.

At this point, we hope that you know already the basics of Theta Suite and how to create and run ThetaML models. There are many other options available, simply play around and get familiar with the software features, it will come handy when later using ThetaML in financial modeling.



Overview of the ThetaML Language

The Theta Modeling Language (ThetaML) is a Domain Specific Language (DSL). ThetaML is an extension to simple procedural programming languages. It is a notation for describing the structure of financial products. The notation is generic, simple and yet is backed by a solid mathematical and computational interpretation. ThetaML is unprecedented in its ability to program in chronological order, above and beyond the conventional computational programming order.

ThetaML focuses on financial product structural features and abstracts these structural features from numerical details. ThetaML has the following features:

- **Expressiveness:** *all features of a financial product are presented in a precise and compact manner.*
- **Modularity:** *the structure of financial products is separated from model stochastics and numerical details.*
- **Transparency:** *ThetaML is easily comprehensible and allows for concise communication and documentation of financial product details.*
- **Integrability:** *ThetaML can be easily integrated with the existing code, which enables the users to build upon previous work.*
- **Simplicity:** *little learning effort is required for a computer literate.*



Why Expressiveness, Modularity, Transparency, Integrability and Simplicity are the most important properties for a Domain Specific Language in finance? The question may be best answered by the following texts:

Expressiveness is important since a description language should ideally be able to encompass all product features of different financial contracts in an investment portfolio. It is also important that all necessary delivery options and day count conventions included in the transactions can be precisely and concisely represented.

Modularity allows separation of responsibilities, such that product structurers can focus on product features and quants can focus on the pricing models. Furthermore, modular code greatly improves maintainability.

Transparency significantly improves program maintainability. ThetaML also describes financial products in an intuitive and compact way and makes it easier to communicate how financial products work.

Integrability with existing functionalities can be guaranteed by the ThetaML programming paradigm which is very close to standard procedural programming that facilitates variables and procedure calls. Reuse of stochastic models and numerical procedures is important to a smooth transition from the previous modeling process to the new one.

The **ThetaML language is simple** and easy to learn. Basic programming skills suffice to learn the language.



3 ThetaML Syntax Reference

This chapter documents the Syntax allowed in ThetaML models. The following contents are included:

- Defining a model
- Using comments
- Assignments
- Model time passing with the `theta` command
- Using `fork ... end` to define parallel processes in time
- `if ... else ... end` statement for conditional evaluations
- Arrays in ThetaML



3.1 Defining a Model

A model in ThetaML is equivalent to a function or a subroutine in other procedural programming languages.

A model in ThetaML has the following structure:

```

model <Model Name>
    <Import/Export Statements>

    <Processing Statements>
end

```

A model starts with the keyword **model** and is terminated with the keyword **end**. The model name follows the keyword **model**.

Model arguments can be parameters or processes and they are imported into the model structure using the keyword **import**. Imported parameters or processes are given a name and an optional description string in double quotes, e.g. "**import variables description string**". The description for the imported variables shows up in the **ThetaML Configurator**³ used to run the model. Multiple parameters or processes can be imported using a single **import** keyword, they are separated by commas.

The values or processes computed by the model are returned in named variables via the keyword **export**. They can be used for further analyses or be imported by other models that call this model as a subroutine. Exported variables must be assigned a value within the body of the model. The exported variables can have an optional description string, e.g. "**export variables description string**".

³ For descriptions on the ThetaML Configurator, please refer to section 2.2 step 5).



The description for the exported variables shows up in the **Theta Suite Result Explorer** ⁴. Multiple parameters or processes can be exported using a single **export** keyword, they are separated by commas.

ThetaML uses the same naming convention for parameters as other programming languages. Variable names can contain letters, digits and single underscore characters; spaces, punctuation marks and symbols are not allowed. Also, variable names cannot start with digits. In ThetaML, variable names are case sensitive. For example, `put` differs from `Put`. Certain ThetaML reserved keywords and type names cannot be used as variable names. The length of variable names is not limited, but it is good programming style to keep variable names short and informative. Examples of valid variable names in ThetaML are:

AmericanOption, discountBond, _p_HestonModel, HW2factor

The following terms are reserved keywords in ThetaML:

`assert, call, else, end, export, fork, from, if, implements, import, inf, interface, length, loop, model, theta, to, type, workflow`

The following terms are reserved type names in ThetaML:

`boolean, date, enum, file, float, object, outputfile, string`

The first letter of each keyword and type name can be capitalized, i.e. `If` and `if` are the same, `Boolean` and `boolean` are the same, and so on.

⁴ For descriptions on the Theta Suite Result Explorer, please refer to section 2.2 step 7).

The following are reserved function names in ThetaML:

`Beta()`, `E()`, `rand()`, `randn()`

The following are reserved operator symbols in ThetaML:

! (the exclamation mark), @ (the at sign)

The following are reserved parameter functions in ThetaML:

`@time`, `@dt`, `@scenarioIndex`, `@scenarioSize`

In ThetaML, the **import** and **export** statements can occur once on a single line or multiple times (in which case, they are separated with a semi-colon ";"). For readability and good programming style, we recommend using one **import** or **export** statement per line. Below is an example for using the **import** and **export** statements:

```
import x1          "stochastic process x1"
import x2          "stochastic process x2"
import a, b, c     "processes a, b, c"

export y1          "stochastic process y1"
export y2          "stochastic process y2"
export e, f, g     "processes e, f, g"
```

In ThetaML, variables are imported as processes. The arguments are imported as if there were passed by reference because the values of a variable can change when model time changes. Stated differently, a process may take different values at different model times in the modeling process. After the **import** and **export** statements, the body of a model processes the code statements that define a model.



Note: In ThetaML, model arguments are imported as processes, they are passed by reference; their values can change when model time passes.

At the time of import, the type of an imported variable is unknown. It is determined later on in any of the following ways:

- In the modeling process, the specific use of the variable implicitly determines its type.
- When imported from an external model, the variable type is determined externally.
- Numeric data types in ThetaML are double-precision floating-point numbers.
- If the other ways are inadequate, the `type` keyword can explicitly specify its type (after all the `import` and `export` statements).

Chapter 4 “The ThetaML type system” gives details on use of variable types in ThetaML.

Note: In ThetaML, numerical data types are double-precision floating-point numbers.

Following are some examples defining models and importing/exporting variables into the models. All code statements in ThetaML can be optionally terminated with the ";" symbol (a semi-colon).

Example 1:

An empty model is formulated like this:

```
1: model EmptyModel
2:
3: end
```

Example 2:

The following model imports a variable x (line 2) and exports its squared value (line 5) in the variable y :

```
1: model xSquared
2:   import x
3:   export y "x squared"
4:
5:    $y = x^2$ 
6:
7: end
```



Example 3:

The following model imports a variable x (line 2) and exports double its value (line 5 - line 7) in the variable y :

```

1: model Double_x
2:     import x
3:     export y

4:     % the value of a! waits to be determined later
5:     y = a! * x

6:     % a is assigned 2, as such a! = 2 at 5
7:     a = 2
8:
9: end

```

In the model `Double_x`, the variable a is referenced by the future operator “!” in line 5. The future operator “!” accesses $a = 2$ at line 7, and returns the result $2 * x$ in the variable y (line 5).



3.2 Using Comments

Comments in ThetaML are initiated with the symbol `%`, everything after the symbol `%` until the end of the line is ignored by the compiler, including comments within comments.

The following is an example of using comments in ThetaML:

```
model Comments
  %this
  % is
  % % a % comment
end
```



3.3 Assignment Operator

Assignments are done using the = operator, e.g.

<variable> = <formula>

Examples of using assignments in ThetaML:

```
x = 42  
x = 7 * 9  
y = 2 + x
```

Note that in the last statement, the variables x and y can be, for example, financial processes with scenario and time as indexes. In ThetaML, processes implicitly incorporate both scenario and time indices. Thus, scalar variables and matrices have the same notation. This simplified notation will come in very handy later when used in solving financial problems.

Note: In ThetaML, a variable that denotes a process implicitly incorporates both scenario and time indexes.



3.4 The `theta` command

The `theta` command is a crucial statement in ThetaML. It defines and passes the model time. Every `theta` statement is followed by a statement defining a time interval.

The `theta` command:

```
theta <time step>  
or  
Theta <time step>
```

Model time is used to synchronize multiple threads that occur parallel in virtual time. When external numerical routines⁵ are called to compute certain model parameters, model time also allows synchronizations with the external numerical routines.

Note: In ThetaML, model time proceeds forward in time according to the timing order of events happening in a pricing model. Model time is passed by the `theta` command.

⁵ The use of external numerical routines in ThetaML is further elaborated in section 3.10 Matlab Native Access



The ThetaML example below illustrates how model time works.

Example 4:

```

1: model A
2:   import S    "S is a process computed in model B"
3:   export X
4:   export Y
5:
6:   theta 0.5    % time passes by 0.5 unit
7:   X = S        % X is assigned the value of S at time 0.5
8:
9:   theta 1.5    % time passes by another 1.5 unit
10:  Y = S        % Y is assigned the value of S at time 2
11:
12: end

```

```

1: model B
2:   export S    "Process S"
3:
4:   S = 1        % at time 0, S = 1
5:   theta 1      % time passes by 1 unit
6:   S = 2        % at time 1, S = 2
7:
8: end

```

When a model that calls or references models A and B is run, the variable `s` is modified and exported by model B, and subsequently used in model A.

In model B, the variable `s` is assigned a value of 1 at time 0 (line 4), and a value of 2 (line 6) at time 1 (1 time unit is passed by the `theta` command). In between time 0 and 1, the variable `s` takes the value of 1 assigned at time 0. After time 1, the variable `s` takes the value of 2 assigned at time 1.



In the model that calls or references models A and B, model A imports the process *s* exported from model B. Since *s* is imported as a process, it automatically synchronizes in model time with the processes *x* and *y* defined in model A.

In model A, as time passes by 0.5 units with the command `theta 0.5` (line 6), the variable *x* is assigned the value of *s* at time 0.5, i.e. $x = 1$ (line 7). As time passes by a further 1.5 units, `theta 1.5` (line 9), the variable *y* is assigned the value of *s* at time 2, i.e. $y = 2$ (line 10).

The example below is simple to understand. Figure 4 illustrates the example's model time passing using the `theta` command.

Example 5:

```

y = 0      % y = 0 at time 0
theta 1    % 1 time unit passes
y = 3      % an event happens at time 1: y is assigned 3
theta 0.5  % another 0.5 time units pass
y = 1      % an event happens at time 1.5: y is assigned 1
theta 1.5  % another 1.5 time units pass
y = 4      % an event happens at time 3: y is assigned 4
theta 1/4  % another 1/4 time units pass
y = 2      % an event happens at time 3 and 1/4:
           % y is assigned 2

```

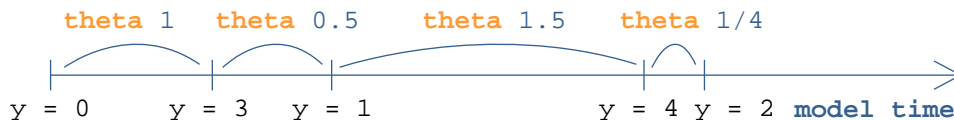


Figure 4. Model time passing with the `theta` command from Example 5.

The model time grid is divided at the time when an event happens. The `theta` command passes model time from one point in time to the next one.



In Figure 4, the variable y is set to 0 at time 0. With time passing by 1 time unit, $y = 3$ at time 1. After another 0.5 time units, $y = 1$ at time 1.5. At time $3\frac{1}{4}$ ($1 + 0.5 + 1.5 + 1/4 = 3\frac{1}{4}$), $y = 2$.



3.5 The `fork ... end` Statement

With the `fork ... end` statement, code blocks are executed in parallel in model time. The `theta` command enables multiple threads bundled by the `fork ... end` statement to be executed in parallel.

The `fork ... end` statement:

```
fork
  commands
end
```

or

```
fork
  commands
end
commands
```

A fork block begins with `fork` and is terminated by the `end` keyword. Statements between these two tokens share the same time axis with other statements in the model, i.e. they run in parallel in model time. Time passing along the time axis is advanced by the `theta` command. A `fork ... end` statement coupled with another `fork ... end` statement run in parallel in model time.

Note: In ThetaML, the `fork ... end` statement enables multiple simulation threads to run (virtually) parallel in model time.



The following example shows two `fork ... end` blocks that run parallel in model time:

Example 6:

```
% the first fork ... end block
1: fork
2:     a = 0      % at time 0, a = 0
3:     theta 2    % 2 time units pass
4:     a = 5      % at time 2, a = 5
5: end

% the second fork ... end block, runs virtually parallel
% with the first fork ... end block
1: fork
2:     theta 1    % 1 time unit passes from time 0
3:     x = a      % at time 1, x = 0
4:     theta 2    % another 2 time units pass
5:     x = a      % at time 3, x = 5
6: end
```

The first block initially sets the variable `a` to zero (line 2), then sets it to 5 (line 4) after 2 time units have passed with the command `theta 2`. Between time 0 and 2, the variable `a` takes the value of 0 assigned at time 0. After time 2, the variable `a` takes the value 5 assigned at time 2.

The second block shares the same model time axis with the first `fork ... end` block. It proceeds to time 1 with the command `theta 1` (line 2), and copies the value of `a` at time 1 (`a = 0`) to the variable `x` (line 3). After a further 2 time units (line 4), at time 3 the value of `a` is copied to `x` again, i.e. `x = 5` (line 5).

In case multiple write operations occur in the same timing model, the value assigned later overwrites the previous value(s). The following example illustrates this.



Example 7:

```

% the fork ... end block
1: fork
2:     a = 0      % at time 0, a = 0
3:     theta 3    % 3 time units pass
4:     a = 5      % at time 3, a = 5
5: end
6:
7: % code statements after the fork ... end block
8: theta 1        % 1 time unit passes from time 0
9: x = a          % at time 1, x = 0
10: theta 2        % 2 time units pass
11: x = a          % at time 3: x = 5

```

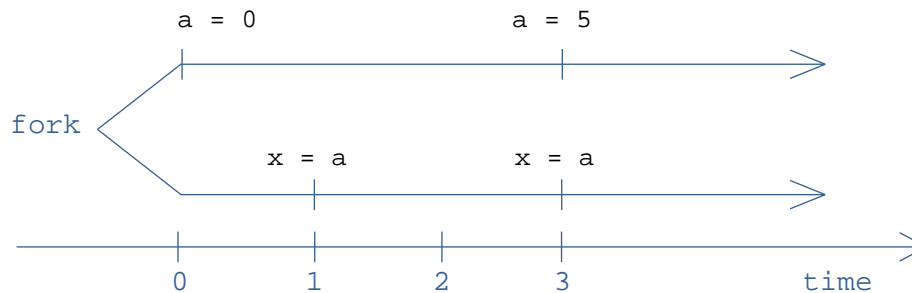


Figure 5. Multiple write operations in the same timing model, from Example 7.

The value assigned to *x* at time 3 overwrites the value assigned to *x* at time 1.

In Example 7, the **fork ... end** code statements (line 1 - line 5) run in parallel with the other code statements in the same model. Within the **fork ... end** code statements, the variable *a* is assigned a value of 0 at time 0 (line 2). At time 3 (line 4), it is given a value of 5. Between time 0 and 3, the variable *a* takes the value 0 assigned at time 0.



The code statements outside the `fork ... end` block share the same model time axis with the variable `a`: as time passes by 1 unit (`theta 1` at Line 8), `x` is assigned the value of `a` at time 1, i.e. $x = 0$ (line 9); after another 2 time units (`theta 2` at line 10), `x` is again given the value of `a` at time 3, i.e. $x = 5$ (line 11). The assignment to `x` at time 3 overwrites the assignment to `x` at time 1, i.e. after time 3, $x = 5$. However, between time 1 and 3, `x` is still equal to 0.

Figure 5 illustrates the code statements in Example 7 in graphical form.

Note: If two ThetaML expressions are defined at the same model time, the coding order of the expressions determines their order in model time.

Another `fork ... end` example:

Example 8:

```

1:      % the first thread
2:      fork
3:          theta 1 % 1 time unit passes from time 0
4:          a = 1   % at time 1, a = 1
5:      end

6:      % a second thread, executed after the first thread
7:      fork
8:          theta 1 % 1 time unit passes from time 0
9:              % at time 1, a = 2, this overwrites
              % a = 1 in the first thread
10:         a = a + 1
11:     end
12:
13:     % a third thread, executed after the second thread
14:     theta 1 % 1 time unit passes from time 0
15:     x = a    % at time 1, x is assigned the
               % value of a = 2

```

In the above example, the three threads run in parallel in model time. The first two threads have different values for the variable `a` at time 1. The second thread (line 7 - line 11) builds upon the latest value of `a` which is `a = 1` (line 4) in the first thread, then increments it by 1 to have the result `a = 2`. A third thread (line 14 - line 15) after the two `fork ... end` blocks could only see the latest value assigned by the second thread (`a = a + 1` at line 10), thus it returns the value of 2 to `x` (line 15).



3.6 The `if ... else ... end` Statement

As with other programming languages, ThetaML allows conditional evaluations using the `if ... else ... end` statement.

The `if ... else ... end` statement:

```
if <condition>
...
else
...
end
```

If the `if` condition evaluates to be true, the statements after `if` and before `else` are executed. Otherwise, the `else` branch is executed.

In ThetaML, the nested `if ... else ... end` statement is as follows:

```
if <condition>
...
else
    if <condition>
        ...
    else
        ...
    end
end
```



ThetaML uses the following logical and relational operators:

```
|  %or
&  %and
>  %greater than
<  %less than
== %is equivalent to
```

The relational operators (e.g. >, <, ==) precede the logical operators (e.g. &, |), the & operator precedes the | operator. Relational operators do only pairwise comparisons, e.g. $a > b$ or $x < y$, but statement like $a > b > c$ is not allowed. In the statement $a > b \& c > d \mid x > y$, the relational comparisons $a > b$, $c > d$ and $x > y$ are evaluated first, then the part $a > b \& c > d$ is evaluated, finally $a > b \& c > d \mid x > y$ is evaluated.

The following examples assign a value to the variable x under certain conditions:

Example 9:

```
...

r = randn() % sample a standard normal random variable

if r < 0 | r > 1 %'or' comparison
    x = 1
else
    x = 2
end

x = 0
if r > 0 & r < 0.1 %'and' comparison
    x = 1
end
```



Example 10:

```
if a > 1 | b < 10 %'or' comparison
    x = 1
end
```

Example 11:

```
if a > 1
    if b < 10
        x = 1
    else
        x = 2
    end
end
```



3.7 Array in ThetaML

We can define an array in ThetaML in several ways.

Arrays can be defined in Matlab⁶ style, using the range operator

```
[from : to]
```

or

```
[from : stepSize : to].
```

Examples:

```
% S is an array of length 5;  
% it ranges from 1 to 5,  
% with increments of 1  
n = 5  
S = [1:n]
```

or

```
% S is an array of length 50,  
% ranging from 1 to 50,  
% with step size 0.5  
n = 50  
S = [1:0.5:n]
```

Array dimensions are fixed at compile time.

Array can also be defined by direct assignment

```
% S is an array of length 4, with values 1,2,3,4  
S = [1, 2, 3, 4]
```

⁶ Matlab is a registered trademark of The MathWorks Inc.



In this case, the compiler automatically detects the length and type of the array `s`.

In cases when the compiler cannot detect the array type (which occurs rarely), array can be declared using the `type` keyword after the `import/export` statements:

```
type stocks float[10]
```

The array `stocks` is of `float` type with size 10. Note the size of the array in the square bracket has to be a literal constant number, such as 10 in the above example.

In ThetaML, array elements are accessed with C-style square brackets `[]` after the array name. Unlike C and like Matlab, ThetaML array indexes start at 1.

Example of array indexing in ThetaML:

```
A = [1, 2, 3, 4]
X = A[1] + A[3]  % A[1] = 1, A[3] = 3,
                  % and X = A[1] + A[3] = 1 + 3 = 4
```

The assignment `=` operator is applicable to array types. An array can be assigned to another array. A scalar value can also be assigned to an array, in which case each element of the array takes the same value as the assigned scalar value.

Examples of array assignment are given below:

Example 12:

```
% A is automatically an array of the same length
% and with the same values as B
B = [1, 2, 3, 4] % the elements of B are float numbers
A = B           % element-by-element array assignment
```



Example 13:

```
% A is an array with length n
A = [1:n] % A is an array of length n
A = 1     % equivalent to A = [1,...,1],
          % i.e. n numbers of 1
```

Example 14:

```
% A is an array of type float with length 4.
% It is explicitly defined using the keyword type after
% the import/export statements

type A float[4] % A is an array of float with length 4

b = 1           % b is of type float
A = b           % equivalent to A = [1, 1, 1, 1]
```

The numeric types are of double precision floating-point numbers in the above examples. In Example 13, if the length of the array `n` is a floating-point number with decimal points, the value of `n` is rounded down to the nearest integer value. For example, if `n` is 10.2, the length of the array `A` is actually 10. If `n` takes the value of 10.8, the length of the array `A` is also 10. The length of an array can be obtained using the `length()` function in ThetaML Script. Details on the `length()` function can be found in section 3.12.3.



3.8 The `loop ... end` Statement

Repeated executions can be achieved using the `loop ... end` statement. The `loop` statement requires a length parameter that must be set at compile time. The parameter defines the type of loops. It can be an integer for a finite number of iterations, or the keyword `inf`, where the loop runs as long as the model is run. The `loop` statement can also take an array as a parameter, in which case the loop will iterate over the elements of the array.

3.8.1 Fixed Length Loop

Fixed length loops are defined by a `loop` command followed by the number of iterations.

The fixed length loop statement:

```
loop <number of cycles>
    ....
end
```

Variables are initiated at the start of the loop.

A fixed `loop ... end` example:

Example 15:

```
1: x = 5           % x is initialized with value 5
2: loop 5          % loop 5 times
3:     x = x + 1    % increment the value of x by 1 at each loop
4: end
```



In Example 15, the variable x is initialized with a value of 5 (line 1), then it goes through a loop 5 times (line 2 - line 4) and is assigned a final value of 10.

The following example uses a fixed loop to update the process y up to time t :

Example 16:

```

1:  t = 1      %time horizon
2:  ht = 0.3   %time step
3:  y = 1      %initial value of y

4:  % a fixed loop of length floor(t/ht), where floor()
5:  % means round down to the nearest integer value
6:  loop t/ht

7:      % time passes by ht time step
8:      theta ht

9:      % update y for the time step ht
10:     y = y * exp( -0.05 * ht )
11: end

12: theta t - @time
13: y = y * exp( -0.05 * @dt )

```

In Example 16, the variable y is initialized at 1 (line 3), it is then updated during the time steps until time t with the fixed loop **loop ... end**. The fixed loop length is the value of t/ht rounded down to the nearest integer value, i.e. 3. The numerator t and the denominator ht are of float type, since in ThetaML numeric types are floating-point numbers. Line 8 passes ht time using the **theta** command. Line 10 updates the value of y for the time step ht . After 3 loops with each loop progresses time by 0.3, the time remaining till time t is 0.1 ($1 - 3*0.3$). Line 12 creates the residual time step which is required to step from the current model time to time t ($t - @time = 1 - 3*0.3 = 0.1$). The **@dt** parameter at line 13 extracts the residual time to the next model time point t , i.e. **@dt** = $t - @time = 0.1$.



3.8.2 Infinite Length Loop `loop inf ... end`

When using `inf` as the loop length, the loop is run until all other threads (sharing the same model time axis) with fixed-length loops have finished their iterations. This is useful when computing time series of an arbitrary length or when writing subroutines for financial products whose lifetime is automatically extended to the desired length depending on a specific pricing application.

The infinite loop statement:

```
% unbounded loop ...
loop inf
    theta @dt % passes an arbitrary time
end
```

or

```
% unbounded loop ...
loop inf
    theta <time interval> % passes explicit time step
end
```

A stand-alone `loop inf ... end` example where the loop runs forever:

Example 17:

```
x = 0          % initialize x with 0
%infinite loop
loop inf
    x = x + 1 % increment the value of x by 1
    theta @dt % passes time by @dt time unit
end
```



An example of `loop inf ... end` where the loop length is determined by a fixed-length loop sharing the same model time axis:

Example 18:

```
% fork ... end block, runs parallel with other code blocks
1: fork
2:   y = 0           % initialize y at 0
3:   % infinite loop, it stops when the fixed loop below stops
4:   loop inf
5:     theta 1       % time passes by 1 unit from time 0
6:     y = y + 1     % increment y by 1 each time
7:   end
8: end

9: % loop 2 times, this loop runs parallel with fork ... end block
10: loop 2
11:   theta 3        % each loop passes time by 3 units
12: end
13:
14: x = y            % time is now at 6, x = 6
```

In Example 18, the `fork ... end` block (line 1 - line 8) runs in parallel with the `loop ... end` block (line 10 - line 12) that follows it. Since the two blocks share the same model time axis, the infinite loop `loop inf ... end` (line 4 - line 7) within the `fork ... end` block runs until the fixed length loop (line 10 - line 12) stop running, in this case, the loop runs 2 times. After 2 loops with each loop passing by 3 time units, both loops stop at time 6. The value of `y` is initialized at 0 (line 2) and is incremented by 1 after each time unit (line 6). After 6 loops, `y` has the value 6. At time 6, `x` is assigned the value of 6 (line 14).

Note: In the same timing model, the infinite loop `loop inf` runs until all other fixed length loops stop running.



3.8.3 Array Looping

Loops can be used to iterate over arrays, much like the *for ... each* statement in many other programming languages. Loops over an array repeat the loop body once for every component of the array. For loops over arrays, the `loop` keyword is followed by a variable that serves as an iterator for each array component, then a colon “:” followed by the array to be cycled. The iterator points directly to the array element, so operations on the iterator are in fact operations on the corresponding element of the array. The keyword `indexOf()` returns the matching index of the element of the array.

The array loop statement:

```
loop array_iterator : array
    % loop body
    index = indexOf(array_iterator)
end
```

An example for a loop over an array:

Example 19:

```
% define the array A
A = [1, 2, 3, 4]

% 'a' serves as an iterator for the array A
loop a : A    % loop through the array A
    a = a^2    % square the value of the array element denoted by a
end           % after the loop, A = [1, 4, 9, 16]
```

In Example 19, the loop cycles over the elements of the array `a` and squares the value of each of its elements. Note that the array `A` is defined in the model body. In cases when an array is imported into the model, the values of the input array can not be changed inside the model.



3.8.4 Multiple Array Looping

Loops in ThetaML can simultaneously cycle over multiple arrays, given that the arrays are of the same length. An iterator is defined for each array so that at the n^{th} cycle of the loop each iterator points to the n^{th} element of its corresponding array. Additionally, it is possible to iterate over a previously undefined array, in which case the size of the undefined array is determined by the size of other arrays within the same loop. This allows the user to build an array on the fly.

The following is an example for loop over multiple arrays.

Example 20:

```

1: % define the array A
2: A = [1, 2, 3, 4]

3: % define the array B
4: B = [1, 2, 2, 2]

5: % loop through the elements of array A and B, and
6: % build a new array X. a, b, x serve as array iterators
7: loop a, b, x : A, B, X
8:     % sum up the values denoted by a and b, and assign
9:     % the value to the element denoted by the iterator x
10:    x = a + b
11: end
12: % after the loop, X = [2, 4, 5, 6]

```

In Example 20, arrays `A` and `B` are both of size 4. The loop cycles over each element of the arrays (line 7). The variables `a` and `b` serve respectively as iterators for arrays `A` and `B`. The variable `x` is previously undefined and is automatically determined to be an array with the same length as arrays `A` and `B`. The elements of `x` take their values in the loop body via the statement `x = a + b` (line 10).



Equivalently, the above loop could be replaced by the statement $x = A + B$. Array assignment, addition, subtraction, multiplication and division are supported in ThetaML. Array multiplication and division are performed element-wise, involving no matrix math.



3.9 Calling A Sub-model

Sub-model refers to the model called by other model(s). Sub-models can be called with a **call** statement in a ThetaML model. When using a **call** statement in ThetaML, the called sub-model must be supplied with all the arguments as required by its **import** statements. Values exported by the sub-model can be optionally imported by the calling model. All arguments in ThetaML are passed similar to passing by reference. Hence not only static values but also processes whose values change in time can be passed to called sub-models.

The **call** command:

```
call <model name>  
    export <local variables>  
    import <remote variables>
```

or

```
call <model name>  
    export <local variables> to <remote variables>  
    import <local variables> from <remote variables>
```



Implicit `fork ... end`

In ThetaML, a model call is implicitly a statement within a `fork ... end` block, i.e.

```
call <model name>
  export <local variables>
  import <remote variables>
```

is equivalent to

```
fork
  call <model name>
    export <local variables>
    import <remote variables>
end
```

The implicit `fork ... end` enables the processes imported from the called model to run virtually parallel in model time with processes in the calling model.

Note: In ThetaML, model call is implicitly within a `fork ... end`.

The following examples show how to call a model in ThetaML.

Example 21:

```
1: % the sub_model is called by another model
2: call sub_model
3:   export a, b           % export the local variables a, b
4:   export a + b to c     % export a + b to the remote variable c
5:   import x, y           % import the remote variables x, y
6:   import z from x + y   % import in the local z from x + y
```

In example 21, the `sub_model` imports (reads) the variables `a`, `b` and `c` and exports (returns) variables `x` and `y`.

When the model `sub_model` is called by another model, the calling model exports its parameter variables `a` and `b` to the corresponding variables `a` and `b` in `sub_model` (line 3). It also exports `a + b` to the variable `c` in `sub_model` (line 4). The calling model in turn imports the variables `x` and `y` returned by `sub_model` (line 5), and also imports (creates) a variable `z` from `x + y` (line 6).

When calling a `sub_model`, if the variables passed by the calling model have different names from those in the `sub_model`, we add the keyword **to** in the **export** statement and the keyword **from** in the **import** statement. Specifically, we **export** the local variables **to** the corresponding variables in the `sub_model`, and we **import** (or create) local variables **from** the variables exported by the `sub_model`.

Example 22:

Suppose now we wish to call a `sub_model` from another model.

```
% the sub_model is to be called later by other models
1: model sub_model
2:     import x "Step size"
3:     export y "Incremented process"
4:
5:     y = 0           % initialize y at 0
7:     loop inf       % infinite loop
8:         theta 1     % time passes by 1 time unit
9:         y = y + x   % increment the process y by a step size x
10:    end
11:
12: end
```

The above `sub_model` imports the variable `x` as a step size (line 2), and uses it to increase the process `y` that is initialized at 0. The `sub_model` returns the result in



the variable y after every passing time unit (**theta** 1 at line 8) and for all units of time (**loop inf**).

In our calling model, we set a step size $x = 1$ (line 1), then call the `sub_model` by exporting x to it (line 5) and read the result back in the variable y by importing from the `sub_model` (line 8).

```
% a calling model that calls the sub_model
1: x = 1 % set the step size x = 1
2: call sub_model % call the sub_model
3:     % export the local variable x to the corresponding x
4:     % in sub_model
5:     export x
6:     % import the values for the local variable y
7:     % from the corresponding y in sub_model
8:     import y
9:
10: % result: y ~ 0 - 1 - 2 - 3 - ...
```

Equivalently, we can call the `sub_model` by directly exporting a value of 1 to x (line 4) and reading the result back in a variable A (line 7) as follows:

```
%a calling model that calls the sub_model
1: call sub_model
2:     % export the value 1 to the variable x
3:     % in sub_model
4:     export 1 to x
5:     % import in (create) the local variable A the values
6:     % from the y returned by the sub_model
7:     import A from y
8:
9:     % result: A ~ 0 - 1 - 2 - 3 - ...
```



Example 23:

This example shows the power of passing processes as model arguments. We can call our `sub_model` once to first create a time series equivalent to either `y` or `A` in example 22, and thereafter use this process as an input argument for the step size of a second process.

```
% call the sub_model to create a process y with
% a time step of 1
call sub_model
    export 1 to x      % export 1 to the variable x in sub_model
    import y          % import the returned values in a local
                      % variable y
%result: y ~ 0 - 1 - 2 - 3 - 4 - 5 - ...

% call the sub_model again to export the just
% created y process as the changing step size for another
% process which is imported as A
call sub_model
    export y to x      % export y to the variable x in sub_model
    import A from y    % import the returned values in a local
                      % variable A

%result: A ~ 0 - 1 - 3 - 6 - 10 - 15 - ...
```



We now show a complete example of a model call in ThetaML.

Example 24:

```

1: model calling_model
2:   export y
3:
4:   % the value of z is imported from sub_model; y = 100
5:   y = z!
6:   call sub_model
7:     % export 10 to x in sub_model
8:     export 10 to x
9:     % import in z the values (100) from y in sub_model
10:    import z from y
11:
12: end

13: model sub_model
14:   import x
15:   export y "x squared"
16:
17:   y = x^2
18:
19: end

```

In example 24, the `calling_model` calls the model `sub_model`, and exports a value of 10 to the `x` in `sub_model` (line 8), then imports in `z` the value of $y = x^2$ (line 10). In the `calling_model`, the future operator “!” accompanying the variable `z` (line 5) accesses the value of `z` imported next from the `sub_model` (line 10), and assigns it to `y` (line 5).



3.10 Matlab Native Access

It is one of the design principles of ThetaML to transparently provide access to the functionalities of the target language. Our main target host language is Matlab. There are two main mechanisms in ThetaML that provide accesses to Matlab:

- Calling Matlab functions
A Matlab function call maps a number of input parameters to one output parameter. It takes the simple form of $y = f(a, b, \dots)$ in ThetaML.

Matlab function calls have the following properties:

- No internal state
 - No global variables allowed
- Calling Matlab complex stepping object
The complex stepping object is implemented in native Matlab m-code. It may have an internal state and usually has a behavior depending on model time.

Because this called complex stepping object steps through model time along with other processes in the calling ThetaML model, this object is also known as **stepping function**.

A complex stepping object has the following properties:

- The object has an internal state.
- The object can import other processes.
- The object can have construction parameters.
- All variables and their stochastic dependencies must be inspectable through the Application Programming Interface (API).



3.10.1 Calling Matlab Functions

Matlab functions can be called in ThetaML by directly typing their function names. Only one variable can be returned by a function and the parameters of the function are only defined in terms of the elements of a vector or matrix.

The following example ThetaML calls a MATLAB function `atan`:

```
x = 1  
y = atan(x)
```

if Matlab is the default backend; otherwise the explicit form

```
x = 1  
y = @matlab : atan(x)
```

must be used.

User-defined Matlab m-files must be vectorized before they are called in ThetaML, the m-files must be located on the Matlab path.

Note: When Matlab functions are accessed in ThetaML, they must be vectorized. This is to ensure smooth handling of different Monte Carlo paths in the first dimension of each variable used in a ThetaML model.



An examples of nonvectorized versus verctorized Matlab functions:

```
% function not vectorized
function result = f_nonvectorized(x)
    result = scalarFunction(x);
end

%simple vectorized function
function result = f_vectorized(x)
    result = zeros(size(x,1), 1);
    for i = 1:size(x,1)
        result(i) = someFunction(x);
    end
end

%function vectorized using Matlab matrices
function result = f_vectorizedMatlab(x, y)
    result = x .* y + randn(size(x));
end
```



3.10.2 Calling a Complex Stepping Object

The link to a Matlab stepping object (stepping function) is made explicitly by calling the Matlab stepping object with a ThetaML command:

```
call @matlab : <stepping function>
      export <local variables>
      import <remote variables>
```

or

```
call @matlab : <stepping function>
      export <local variables> to <remote variables>
      import <local variables> from <remote variables>
```

The called stepping object allows many sophisticated interactions but it must implement in Matlab the methods listed in Table 1.

Method	Description
Init()	Reset the model to its initial state (this method is optional)
Step(dt)	Proceed the model by time dt. Throw random numbers if required
SetValues(name, value)	Set a state variable with name to the new value
GetValues(name)	Get the value of a variable name computed by this model
GetModelVariables()	Give a list of imported and exported variables for this mode

Table 1. Methods for the Matlab stepping object.
Column 1 lists the methods that must be implemented for the complex stepping object, column 2 gives a short description for the methods.

The general function header for the Matlab stepping object consists of method definitions and a construction from a Parameters object. This type of object interactions only works for Monte-Carlo evaluations. The structure of the Parameters object is guaranteed to contain at least one field: NoOfScenarios, with the number of Monte-Carlo paths as its parameter value.



Below is a general function header for the Matlab stepping object:

```
% Function header for the complex stepping object.
% The function takes a 'Parameters' object as argument,
% and returns a struct 'model'
function model = ModelName(Parameters)

% The following struct fields must be implemented in this
% function and must return the function handles with '@'
% notation.
model.Step                = @Step;
model.SetValues           = @SetValues;
model.GetValues           = @GetValues;
model.GetModelVariables   = @GetModelVariables;
model.Init                = @Init; % (optional)

if nargin > 0
    %construct the object
else
    %construct an empty object for import variable inspection
end
```



The model variable(s) returned by the `GetModelVariables` method must be a structure with fields for each relevant variable. Each field must have, for each variable, the sub fields listed in Table 2.

comment	A human readable description of the roles of the variables in the model
Visibility	The flag <code>Visibility</code> can be set to import or export , it determines whether the variable is imported as an input argument or exported as an output variable returned by the model
IsState	For export variables, this Boolean flag indicates whether the variable is part of the minimal Markov state which is required to make a best guess for the future value of all exported variables
Size / Type	<p>The flag <code>Size</code> indicates the dimension of a model variable, excluding the scenario dimension.</p> <p>For non-numeric types[†], the <code>Type</code> field can be set to <code>'Object'</code> or <code>'String'</code>⁷. Import variables can also have their <code>Type</code> set to <code>'File'</code>⁸ which indicates that the value must be the Uniform Resource Identifier (URI) of an existing file</p>

Table 2. Sub-Structure Fields for the `GetModelVariables` Method

The sub-structure fields are listed in column 1, their respective descriptions are given in column 2.

[†] For details on variable types in ThetaML, please refer to Chapter 4.

⁷ `Object` and `String` are ThetaML internal type.

⁸ `File` is a ThetaML type.



To guarantee the correctness of the generated code, it is critical that the internal state of the object can be inspected as relevant. All variables that can be used to infer the stochastic properties of the variables must be made explicit. Additionally, variables can be marked as being part of the minimal Markov state by setting `IsState = true`. By minimal Markov state we mean that the variables satisfy the Markov property and that with minimal amount of representative information, it is sufficient to make best guesses for the future state. A `visibility` setting can hide a variable from its explicit use in ThetaML.

Below, there are some examples for Matlab stepping functions and use of the Matlab stepping functions in ThetaML.



Example 25:

We start with a simple option pricing example that uses a Matlab stepping function to short-step the stock price process. Copy the following model into a new Matlab file and save it under the name `GBMModel.m`.

```

1: % The Matlab function GBMModel takes a 'param' object as
2: % argument, and returns a Matlab struct 'model'
3: function model = GBMModel(param)
4: % The following struct fields must be implemented in this
5: % function and must return the function handles with '@'
6: % notation.
7: model.Step = @Step;
8: model.SetValues = @SetValues;
9: model.GetValues = @GetValues;
10: model.GetModelVariables = @GetModelVariables;
11: model.Init = @Init; % optional function
12:
13: % if the number of function arguments is larger than zero
14: if nargin > 0
15:     % remember initial values of imports stored in 'param'
16:     model.vola = param.vola; %stock price volatility 'vola'
17:     model.r = param.r; %risk-free interest rate 'r'
18:     model.S0 = param.S0; %initial stock price 'S0'
19:
20:     % prepare initial values for exports
21:     % initialize stock prices
22:     model.S = param.S0 * ones(param.NoOfScenarios,1);
23:     % initialize discount factors
24:     model.Discount = ones(param.NoOfScenarios,1);
25: end
26:
27: % this function returns a struct 'vars' with fields and
28: % subfields for all relevant variables
29: function vars = GetModelVariables
30: % create a struct for 'S', set: the field value for
31: % 'comment' to 'Stock price', the field value for
32: % 'Visibility' to 'Export', and the field value for
33: % 'IsState' to 'true'. 'S' is itself a field of the

```



```

32: % struct 'vars'
33: vars.S = struct('comment', 'Stock price', ...
34:                'Visibility', 'Export', ...
35:                'IsState',true);
36: % create a struct for 'Discount'
37: vars.Discount = struct('comment', ...
38:                       'Discount process', ...
39:                       'Visibility', 'Export', ...
40:                       'IsState',false);
41: % create a struct for 'vola'
42: vars.vola = struct('comment','vola', ...
43:                   'Visibility','Import');
44: % create a struct for 'S0'
45: vars.S0 = ...
46:   struct('comment','S0','Visibility','Import');
47: % create a struct for 'r'
48: vars.r = struct('comment','Interest rate', ...
49:                'Visibility', 'Import');
50: end
51: % set the value of 'var' to 'value'
52: function SetValues(var, value)
53:     model.(var) = value;
54: end
55: % get the values of 'varName'
56: function X = GetValues(varName)
57:     X = model.(varName);
58: end
59:
60: % this function steps the process S for the time step dt
61: function Step(dt)
62: % Geometric Brownian motion for stock price process 'S'
63:     model.S = model.S .* ...
64:         exp( (model.r - 0.5 * model.vola^2) * dt...
65:             + sqrt(dt) * model.vola * ...
66:             randn(size(model.S)));
67:
68:     % constant interest rate 'r' as the decaying rate
69:     model.Discount = model.Discount * exp(- model.r * dt);
70: end
71:
72: end

```



In the Matlab function `GBMModel`, the function header consists of the name of the function `GBMModel`, the input argument object `param`, and the returned struct `model`.

Line 5 - 9 define the fields of the struct `model`. The fields of the `model` are embedded functions defined next in the function `GBMModel`. The function `@Init` at line 9 is optional.

The `if` block in line 12 - 23 initialize the parameter values if the number of function arguments is larger than zero.

The embedded function `GetModelVariables` (line 27) takes no arguments and returns a struct of variables `vars`. The first state variable `s` in the struct `vars` has subfields: `comment` with value `'Stock price'`, `Visibility` with value `'Export'`, and `IsState` with value `true`. The parameter `Discount` is also a struct, it has subfields: `comment` with value `'Discount process'`, `Visibility` with value `'Export'`, and `IsState` with value `false`. The constants `vola`, `r` and `S0` are structs with similarly valued fields. The subfield `Visibility` has value `'Import'` or `'Export'` depending on whether the variable is an import parameter or an export parameter. If the subfield `IsState` of a variable is set to the value `true`, the variable is part of the minimal Markov states. In this function, the state variable `s` is marked as part of the minimal Markov states which are used to make best estimates about the future state.

The embedded function `SetValues` sets the state variable named `var` to `value`. The state variable `var` can take many values.

The embedded function `GetValues` returns the values of `varName`.

The embedded function `Step` progresses the stock prices `s` and updates the discounting process `Discount`, for a time step of `dt`. It is this function that enables



the `struct model` to have a behavior depending on model time when called by a ThetaML model.



Next, we call the GBMModel.m in a ThetaML model that prices an Asian option; the ThetaML model is given below:

```

% This ThetaML model prices a fixed strike Arithmetic Asian
% Call option. It calls the Matlab stepping function
% GBMModel to short-step the stock price process and the
% discount factors
1: model callGBM
2: % This model exports a stock price process 'S', a discount
3: % factor process 'CUR', and the simulated Asian Call option
4: % prices 'AsianOption_CUR'
5:   export S      "Stock price process"
6:   export CUR    "Discount process denominated in currency CUR"
7:   export AsianOption_CUR "Present values of an Asian option"
8:
9:   % call the Matlab stepping function GBMModel
10:  call @matlab:GBMModel
11:    export 0.4 to vola % export the value 0.4 to 'vola'
12:    export 100 to S0   % export the value 100 to 'S0'
13:    export 0.05 to r   % export the value 0.05 to 'r'
14:    % import in the local variable 'S' the values of the
15:    % remote variable 'S' returned in the function
16:    % GBMModel
17:    import S from S
18:    % import in the local variable 'CUR' the values of
19:    % the remote variable 'Discount' returned in the
20:    % function GBMModel
21:    import CUR from Discount
22:
23:    Average = 0 % initialize the arithmetic 'Average' to 0
24:    loop 10      % loop 10 times
25:      theta 1/10 % time passes by '1/10' time units
26:      % update the arithmetic 'Average'
27:      Average = Average + S/10
28:    end
29:    % Asian Call option payoffs, discounted to time 0 by
30:    % the discount factors 'CUR'
31:    AsianOption_CUR = max(Average - 100,0) * CUR
32: end

```



After running this model in Theta Suite with the number of Monte-Carlo paths set to 10,000, we obtain the mean value 11.243 for `AsianOption_CUR`.

In the model `callGBM`, line 5, line 6 and line 7 respectively exports the stock price process `S`, the discount factor process `CUR`, and the simulated Asian Call option prices `AsianOption_CUR`.

In ThetaML, model calls implicitly create a `fork`. i.e., the called model bodies are executed virtually parallel with other code blocks in model time. Calls of external Matlab models implemented with the `Step(dt)`⁹ method has a model time behavior, which means that it steps through model time along with other processes in the calling ThetaML model. In this case, the Matlab stepping function `GBMModel` has an embedded `Step(dt)` function (line 61 in the function `GBMModel`). The `Step(dt)` function is interpreted by the ThetaML compiler as if a `theta @dt` command were passed. The `theta @dt` command enables model time passing at an arbitrary time interval, meaning the parameter `@dt` extracts the time interval to the immediate next model time point.

From the called Matlab function `GBMModel`, we import the stock price process in `S` (line 17) and the discount factor process in `CUR` (line 21).

Next, line 23 to 28 computes an arithmetic average `Average` based on the stock prices on a set of fixed times (1/10, 2/10, ..., 1), spaced at constant time interval 1/10.

At option maturity time (line 31), the fixed strike Asian Call option payoffs are defined and discounted to time 0 using the discount factors `CUR`. Discounting future cash flows to time 0 is a convention we very often use in ThetaML models, so that we always talk about future cash flows in present value terms.

⁹ For a description of the `Step(dt)` method, please refer to section 3.10.2.

Example 26:

Another example Matlab stepping function ExternalModule is shown below:

```

% The function ExternalModule takes a 'param' object as
% argument, and returns a Matlab struct 'model'
1: function model = ExternalModule(param)
2:
3:     % the following are fields for the struct 'model'
4:     model.GetModelVariables = @GetModelVariables;
5:     model.GetValues         = @GetValues;
6:     model.SetValues         = @SetValues;
7:     model.Step               = @Step;
8:     % initialize relevant parameter values
9:     if nargin > 0
10:         model.vola = param.vola;
11:         model.Discount = param.Discount;
12:         model.S0 = param.S0;
13:         model.S = param.S0*ones(param.NoOfScenarios,2);
14:     end
15:     % this function returns a struct 'vars' with fields
16:     % and subfields for all relevant variables
17:     function vars = GetModelVariables
18:         % S has a description string 'Stock price'
19:         vars.S.comment = 'Stock price';
20:         % S is exported as output and is visible to the
21:         % calling ThetaML model
22:         vars.S.Visibility = 'Export';
23:         % S is marked as part of the minimal Markov state
24:         vars.S.IsState = true;
25:         % S has two dimensions: the first dimension is the
26:         % number of Monte-Carlo Scenarios, the second is the
27:         % number of stocks
28:         vars.S.Size = 2;
29:         % create struct fields for 'S2'
30:         vars.S2.comment = 'Double Stock price';
31:         vars.S2.Visibility = 'Export';
32:         vars.S2.IsState = false;

```




```

33:         vars.S2.Size = 2;
34:         % vola is an imported parameter, and is visible to the
35:         % calling ThetaML model
36:         vars.vola = struct('comment','vola', ...
37:             'Visibility','Import');
38:         vars.S0 = struct('comment','S0', ...
39:             'Visibility','Import');
40:         vars.Discount = struct('comment','Discount', ...
41:             'Visibility','Import');
42:     end
43:     % sets the value of state variable 'var' to 'value'
44:     function SetValues(var, value)
45:         model.(var) = value;
46:     end
47:     % gets values for the argument string 'str'
48:     function X = GetValues(str)
49:         % if the string argument 'str' is the same as 'S'
50:         if strcmp(str,'S')
51:             % assign the discounted prices of S to X
52:             X(:,1) = model.S(:,1) .* model.Discount;
53:             X(:,2) = model.S(:,2) .* model.Discount;
54:         elseif strcmp(str,'S2')
55:             X = 2*GetValues('S');
56:         end
57:     end
58:     % this function steps the process S for the time step dt
59:     function Step(dt)
60:         model.S = model.S + param.vola .* sqrt(dt)...
61:             *randn(size(model.S));
62:     end
63:
64: end

```

In the Matlab function `ExternalModule`, the function header consists of the name of the function `ExternalModule`, the input argument object `param`, and the returned struct `model`.

Line 4 - 7 define the fields of the struct `model`. The fields of the `model` are embedded functions defined next in the function body.



The embedded function `GetModelVariables` (line 17) takes no arguments and returns a struct of variables `vars`. The first state variable `s` in the struct `vars` has subfields: `comment` with value `'Stock price'`, `Visibility` with value `'Export'`, `IsState` with value `true`, and `Size` with value 2. The second state variable `s2` in the struct `vars` has subfields: `comment` with value `'Double Stock price'`, `Visibility` with value `'Export'`, `IsState` with value `false`, and `Size` with value 2. The constants `vola` and `s0` are structs with similarly valued fields. The parameter `Discount` is also a struct and is imported as a process parameter. The subfield `Visibility` has value `'Import'` or `'Export'` depending on whether the variable is an import parameter or an export parameter. If the subfield `IsState` of a variable is set to the value `true`, the variable is part of the minimal Markov states. In this function, the state variable `s` is marked as part of the minimal Markov states, while the state variable `s2` is not. This is because the double stock price `s2` contains virtually the same amount information as what is already in `s`. As such, including `s` only is enough to make best guesses about the future state.

The embedded function `SetValues` sets the state variable named `var` to `value`. The state variable `var` can take many values. The embedded function `GetValues` is a simple recursive function, the `if` condition compares the string argument `str`, if it evaluates to `'s'`, the function returns a two-dimensional stock prices `s`. If it evaluates to `'s2'`, the function calls itself (line 55) and returns a two-dimensional double stock prices `s2`.

The embedded function `Step` progresses the stock prices `s` for a time step of `dt`. It is this function that enables the struct `model` to have a behavior depending on model time when called by a ThetaML model.

The stepping function `ExternalModule` is called in ThetaML by the command

```
call @matlab : ExternalModule
```



The ThetaML model `packOrder` illustrates the use of the Matlab stepping function

`ExternalModule`:

```

1: % The ThetaML model 'packOrder' calls the Matlab stepping
2: % function 'ExternalModule'. The processes 'S' and 'S2' in
3: % the called 'ExternalModule' runs virtually parallel with
4: % other processes in the calling ThetaML model 'packOrder'
5: model packOrder
6: % This model exports the values of a stock portfolio 'V', a
7: % discount factor process 'Discount', and stock prices 'S'
8: import r "Constant interest rate"
9: export V, Discount, S
10:
11: % a fork ... end block
12: fork
13: % initial value of the discount factors
14: Discount = 1
15: % infinite loop
16: loop inf
17: % time passing of @dt time interval
18: theta @dt
19: % update the discount factors
20: Discount = Discount * exp( -r * @dt )
21: end
22: end
23: % call the Matlab function 'ExternalModule'
24: call @matlab : ExternalModule
25: export Discount % export the Discount factor process
26: export 100 to S0
27: export 0.4 to vola
28: import S2, S % import the processes S2 and S
29: % a fixed loop of length 10
30: loop 10
31: % time passing of 1 time unit
32: theta 1
33: % the values of V! wait to be decided later
34: V = 0.5 * E(V!) + 0.5 * S
35: end

```



```

32:      % V has the values of S2, V! at line 30 references this V
33:      V = S2
34:  end

```

In the model `packOrder`, line 4 imports a constant interest rate `r`. Line 5 exports the values of `V`, the discount factors `Discount`, and stock prices `S`.

The `fork ... end` block (line 8 - line 18) runs in parallel with the other processes outside the `fork` block, i.e. the called external Matlab model (line 20 - line 24) and the fixed-length loop `loop ... end` (line 26 - line 31).

Calling the Matlab function `ExternalModule` implicitly creates a `fork`. The function `ExternalModule` has an embedded `Step(dt)` function that steps forward the stock price process and the discount factor process in model time along with other processes in the model `packOrder`.

The implicit `fork` around the fixed loop `loop ... end` enables the processes inside the loop run virtually parallel in model time with other processes.

Within the `fork ... end` block, the discount factor `Discount` is initialized at 1, then it enters an infinite loop `loop inf` and is discounted at a constant rate `r` at each time step `@dt`.

The infinite loop `loop inf` runs until the fixed loop (line 26 - line 31) stops running, i.e. it runs 10 times as well. The time interval parameter `@dt` extracts the model time interval 1 as passed by the `theta` command (line 28) within the fixed-length loop. The ThetaML compiler passes as well this time interval of length 1 to the `dt` parameter of the `Step(dt)` function in the called Matlab model `ExternalModule`. The called `ExternalModule` is supplied with the local process `Discount`, a value of 100 to `S0`, and a value of 0.4 to `vola`. The ThetaML model `packOrder` imports the two processes `S` and `S2` from `ExternalModule`.



The fixed loop `loop ... end` (line 26 - line 31) repeats 10 times, each loop passes the time by 1 time unit (`theta 1` at line 28). At each passing time step, the variable `v` is updated at line 30. The variable `v!` at line 30 is referenced with the future operator “!”. The future operator “!” enables `v` at line 30 to access its future values. To evaluate the variable `v`, we start from line 33, where $v = s_2$ at time 10 (10 loops with each loop passing by 1 time unit). Then we go backwards and update the values of `v` iteratively. Since there is no time passing between line 30 and line 33, the variable `v!` at line 30 evaluates to the values of `s2` at line 33, the values for `v` at time 10 are updated as: $v = 0.5 * E(s_2) + 0.5 * s = 0.5 * (s_2 + s)$. We have this equality because `s2` is known at time 10, as such $E(s_2) = s_2$. At time 9, the variable `v!` evaluates to the values of `v` at time 10 (i.e. $v = 0.5 * (s_2 + s)$), the values for `v` at time 9 is: $v = 0.5^2 * E(s_2 + s) + 0.5 * s_2$, where $E(s_2 + s)$ is the best guess of `s2 + s` conditional on the information at time 9. Continuing backwards, we arrive at the time 0 the values of `v`.

Since in ThetaML process variables implicitly incorporate scenario and time indices, the variables `s`, `s2`, `Discount` and `v` contain values for all the Monte-Carlo paths, as such we talk about the values of `s`, `s2`, `Discount` and `v` in plural forms.

Note: The Matlab stepping function steps through model time along with the other processes in the calling ThetaML model.

3.11 ThetaML Operators

The Future Operator “!”

A unique feature of ThetaML is the ability to access the values assigned to variables at future time points. This is achieved with the future operator “!”¹⁰.

Like other programming languages, ThetaML accesses the value of a variable that is pre-assigned. In cases where the code statements are not evaluated sequentially, ThetaML can access ahead the value of a variable assigned at future time points. Access to the future value of a variable is enabled in ThetaML by the future operator “!”.

Example 27:

The following example assigns the next value of $y = 3$ to x . Any command that changes the value of y also changes the value of x .

```
% the future operator “!” acts like a function on y,
% as such the value of y! waits to be determined later
x = y!
...
y = 3 % y has the value 3, and y! = 3
```

¹⁰ Note that circular definitions are not allowed in ThetaML.



Example 28:

In some cases, no single instance can determine the value of a future-referenced variable. The value of the referenced variable is then determined as if the program had been run in reversed command order.

```

1: x = y!           % result: x = 0 if a > 1, x = 2 otherwise
2: if a > 1
3:     y = 0         % if a > 1, y = 0, and y! = 0
4: end
5: y = 2           % if a <= 1, y = 2, and y! = 2

```

When y is future-referenced with the future operator “!” ($y!$ at line 1), its value is temporarily undermined. The future operator “!” looks into the future instance when y has some definite value, this occurs at two instances: within the **if** block (line 3) and after the **if** block (line 5). If the **if** condition evaluates to be true, $y = 0$; otherwise, $y = 2$. The variable y then takes the correct value and assigns the value back to x .

Example 29:

A special case occurs when a future-referenced variable is evaluated as part of the **if** conditions. In such cases, future references consider only value assignments after the **if ... end** block. This avoids cyclical definitions of variables.

```

1: x = y!           % x = 0
2: if y! > 1
3:     y = 0
4: end
5: y = 2           % y has the value 2, and y! = 2 in the if condition

```

In the above example, the value of $y!$ (in line 1) waits to be determined. The $y!$ as part of the **if** condition (line 2) accesses its next value of y , which is $y = 2$ after

the `if ... end` block (line 5). Since $y! = 2$ in the `if` condition statement, the `if` condition evaluates to be true, and the body of the `if ... end` block is executed, i.e. $y = 0$ (line 3). As a result, the first $y!$ access this value 0 and assigns x the value 0.

Below are some additional properties of the future operator “!”:

```

a!! == a!
(a + b)! == a! + b!
f(a)! == f(a!)
a[i]! == a![i] (not a![i!])
E(a) == E(a!)
Beta(a,b) == Beta(a!,b!)

```



3.12 Functions

3.12.1 The Function $\mathbb{E}(\cdot)$

ThetaML evaluates the conditional statistical properties of variables or processes with ease and speed. This is realized with the function $\mathbb{E}(\cdot)$. The function $\mathbb{E}(\cdot)$ computes the conditional expected value of a variable, a process or an expression, conditional on all parameter values that are known at the corresponding model time. The arguments of the stochastic function are assumed able to access their future values assigned at future time points.

We use a simplified example to explain the $\mathbb{E}(\cdot)$ function in ThetaML. Assume

$$y(t_{i+1}) = b_1 x(t_i)^1 + b_2 x(t_i)^2 + b_3 x(t_i)^3 + \epsilon(t_{i+1}),$$

where, for example, $y(t_{i+1})$ is the value of a security based on the financial variable x , and $x(t_i)$ is the price of a financial variable at time t_i . The coefficients b_1 , b_2 and b_3 are constant. The terms $x(t_i)^1$, $x(t_i)^2$ and $x(t_i)^3$ are the price for the financial variable at time t_i , respectively to the power of 1, 2 and 3. The term $\epsilon(t_{i+1})$ is a Gaussian random variable.

The $\mathbb{E}(\cdot)$ function computes the expected value of the above formulation $E(y(t_{i+1}) | \sigma(x(t_i))) = b_1 x(t_i)^1 + b_2 x(t_i)^2 + b_3 x(t_i)^3$, conditional all paths of the financial variables $x(t_i)$ at time t_i . The term $\sigma(x(t_i))$ denotes the smallest sigma field of the variable x at time t_i .

Graphically, the relationship between $E(y(t_{i+1}) | \sigma(x(t_i)))$ and $x(t_i)$, estimated with linear regression method for the above formulation is given in Figure 2.

The $\mathbb{E}(\cdot)$ function in ThetaML uses similar idea for estimating the relationship between $E(y(t_{i+1}) | \sigma(x(t_i)))$ and $x(t_i)$ at time t_i , only that it uses advanced numerical algorithms optimized for more efficient and accurate results.



Example 30:

This example computes the variable x as the expected value of $y!$ ($E(y!)$ at line 4), conditional on the information known at time 5. The term $y!$ at line 4 takes the next value of y at time 10 which is $y = S$ at line 6.

```

1:  theta 5      % 5 time units pass
2:  % x is set to the best guess value of y! conditional on
3:  % time 5 information
4:  x = E(y!)
5:  theta 5      % another 5 time units pass
6:  y = S        % at time 10, y has the value of S at time 10

```

The future operator “!” accompanying y at line 4 can be omitted due to the definition of the $E()$ function, i.e.

$$E(y!) == E(y)$$



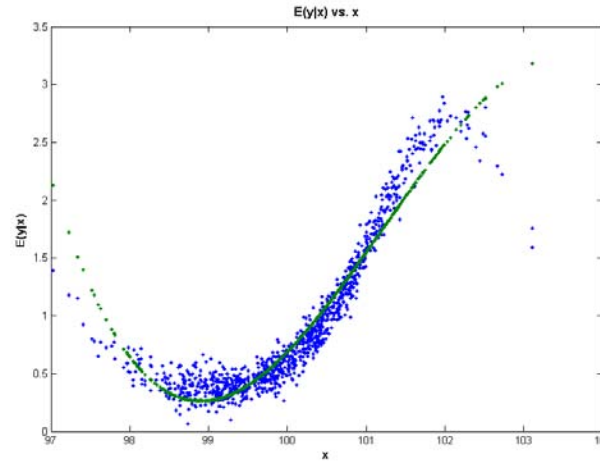


Figure 2. The relationship between $E(y(t_{i+1}) | \sigma(x(t_i)))$ and $x(t_i)$ at time t_i . The term $\sigma(x(t_i))$ denotes the smallest sigma field of the variable x at time t_i .

The variable $x(t_i)$ is created as $x(t_i) = 100 e^{0.01 * \epsilon(t_i)}$, the variable $y(t_{i+1}) = e^{\sin x(t_{i+1})} + 0.1 * \epsilon(t_{i+1})$, where $\epsilon(t_{i+1})$ is the standard normal variable sampled for 1000 paths. The graph is based on the equation $E(y(t_{i+1}) | \sigma(x(t_i))) = b_1 x(t_i)^1 + b_2 x(t_i)^2 + b_3 x(t_i)^3$. In the graph above, the blue colored dots are sample values for the variable $x(t_i)$, the green line is the function value for $E(y(t_{i+1}) | \sigma(x(t_i)))$. This graph is generated in Matlab.

3.12.2 The Function `Beta()`

The `Beta()` function takes two arguments and computes the beta factor(s) between these two arguments, conditional on the current information. The first argument as the explanatory variable(s) can have multiple dimensions in which case the `Beta()` function computes a beta factor for each component of that array. The second argument is the dependent variable for the conditional regression. The conditional regression is enabled with the future operator “!”, since conditional regression involves future values of financial variables unknown at current time and the future operator “!” allows access to future values.

Mathematically, the `Beta()` function is defined as

$$\beta(S(t_{i+1}), V(t_{i+1}) | \sigma(S(t_i))) = \frac{\text{cov}(V(t_{i+1}), S(t_{i+1}) | \sigma(S(t_i)))}{\text{var}(S(t_{i+1}) | \sigma(S(t_i)))}$$

where, the term $V(t_{i+1})$ is the dependent variable, $S(t_{i+1})$ is the explanatory variable. The operator $(|)$ denotes conditioning. The numerator $\text{cov}(V(t_{i+1}), S(t_{i+1}) | \sigma(S(t_i)))$ computes the covariance between the variables $V(t_{i+1})$ and $S(t_{i+1})$, conditional on $\sigma(S(t_i))$ - the time t_i smallest sigma field of $S(t_i)$. The denominator $\text{var}(S(t_{i+1}) | \sigma(S(t_i)))$ computes the variance of $S(t_{i+1})$, conditional on time t_i smallest sigma field $\sigma(S(t_i))$.



3.12.3 Other Functions

The length() Function

The length of an array can be determined using the `length()` function.

Example 31:

```
A = [1, 2, 3, 4] % A is an array with elements 1, 2, 3, 4
L = length(A)    % result: L = 4
```

Example 32:

```
% S is an array of 2 stocks in S
L = length(S)    %result: L = 2
```



3.13 System Parameters

System parameters are compile-time parameters, they can be extracted with a @ sign followed by the keywords `dt` or `time`.

3.13.1 The Parameter @dt

The synchronous time interval parameter `@dt` has different values depending on its context.

If `@dt` is found following the `theta` command, it evaluates to the time interval of the next smallest time step. If `@dt` is located elsewhere, it evaluates to the time elapsed since the thread's previous invocation of the `theta` command. The parameter `@dt` is most often used within the infinite loop `loop inf ... end`.

Example 33:

This example simulates a standard Brownian motion process for all time steps; the parameter `@dt` is an argument to the `theta` command.

```

1: model BrownianMotion
2:
3:     export W "Standard Brownian Motion process"
4:
5:     W = 0      % initialize the W process at 0
6:     loop inf   % infinite loop
7:         theta @dt % passes time by @dt time interval
8:         W = W + sqrt(@dt) * randn() % update the process W
9:     end
10:
11: end

```



In the example model `BrownianMotion`, line 5 initializes the Brownian Motion process `w` at 0, the process then enters an infinite loop `loop inf`, the infinite loop runs until all other fixed-length loops sharing the same model time axis stop running. This happens when the model `BrownianMotion` is called or the exported `w` process is imported by other product models as input parameters. After each time passing of `@dt` units with the `theta` command at line 7, the process `w` is then incremented by a random amount of `sqrt(@dt) * randn()`, where both `sqrt()` and `randn()` are math functions.

The time interval parameter `@dt` in this simulation model remains to be determined. When the model `BrownianMotion` is called by some pricing models with explicit time intervals, the parameter `@dt` then extracts the smallest of all the time intervals from the current model time to the immediate next model time point.



Example 34:

This example illustrates the use of several ThetaML commands: `fork ... end, loop inf ... end, theta @dt`, and `@dt`.

```

1: model SimpleAnnuityModel
2: % This model computes the price of an annuity with
3: % constant interest rate 'r'
4:   import r          "Risk-free interest rate"
5:   import n          "Number of years"
6:   export D          "Discount process"
7:   export Annuity    "Annuity value"
8:
9:   %a fork ... end block
10:  fork
11:    D = 1 % initialize the discount factor at 1
12:    loop inf % infinite loop
13:      theta @dt % theta time passing
14:      D = D * exp(-@dt * r) % update the discount process
15:    end
16:  end
17:
18:  Annuity = 0 % initialize the Annuity value at 0
19:  loop n % fixed loop that loops for n times
20:    theta 1 % theta time passing of 1 year
21:    Annuity = Annuity + 100 * D % update the Annuity value
22:  end
23:
24: end

```

In the model `SimpleAnnuityModel`, line 4 and line 5 import the constant parameters `r` and `n` into the model, line 6 exports the returned discount process `D` and line 7 exports the computed value for `Annuity`.

The body of the model starts from line 10 and ends at line 22. The `fork ... end` code block (line 10 - line 16) runs in parallel with the `loop ... end` code block



(line 18 - line 22), synchronized by the model time passed by the two `theta` commands. It is as if the `loop ... end` block is forked by the `fork ... end` statement as well.

The length of the loop `loop inf ... end` is determined by the length of the loop `loop ... end`, which is n . Since the `theta` command synchronizes the two threads – `loop inf ... end` and `loop ... end`, the time interval parameter `@dt` (in the infinite loop `loop inf ... end`) extracts the time interval passing to the next model time point. The model time grid is determined by the `theta 1` command in the fixed loop `loop ... end`, i.e. at each model time an Annuity of 100 is paid and discounted to the present. At year 10, the Annuity stops. As such, model times in this example are a set of times in years $\{0, 1, 2, \dots, 10\}$, regularly spaced at an interval of 1. Each year, an event happens, i.e. a discounted cash inflow of 100 to the Annuity product.



3.13.2 The Parameter @time

This `@time` parameter provides the current model time. It is the sum of all previous `theta` time steps. The parameter `@time` is most often used within the infinite loop `loop inf ... end`.

Example 35:

```

1: model DiscountFactor
2: % This model computes a discount factor in two ways
3: import r "Risk-free interest rate"
4: export Discount_1 "Discount factor process 1"
5: export Discount_2 "Discount factor process 2"
6:
7: Discount_1 = 1 % at time 0, initialize Discount_1 at 1
8: Discount_2 = 1 % at time 0, initialize Discount_2 at 1
9: loop inf % infinite loop
10: theta @dt % theta time passing at @dt time interval
11: % update Discount_1 for the time interval @dt
12: Discount_1 = Discount_1 * exp( -r * @dt )
13: % update Discount_2 for the time that has passed since
14: % time 0, this is summed by @time
15: Discount_2 = exp( -r * @time )
16: end
17:
18: end

```

The model `DiscountFactor` simulates two equivalent discount factors with constant interest rate r . The parameter `@dt` is the time interval parameter, and `@time` is the sum of all previous time steps `@dt`. For example, if we are at the n -th loop, `@time` would be equivalent to $n * @dt$ if the time intervals extracted by `@dt` are constant intervals. Since the interest rate is constant throughout the model life time, we have used the same r for different discounting time intervals.



3.14 Chapter Example

A First Application of ThetaML to Pricing Financial Derivatives

This example shows how to price an American put option in ThetaML. The American put option is written on a single stock price s , the option maturity is τ , and the option strike price is κ . The put option is early exercisable.

The underlying stock price 's' is assumed following a Geometric Brownian motion process under the risk-neutral measure \mathbb{Q} :

$$dS_t = rS_t dt + \sigma S_t dW_t,$$

where the term S_t is the stock price at time t . The growth rate r and volatility σ of the stock price are constant parameters. The term dW_t are the increments of a standard Brownian motion process under the measure \mathbb{Q} . The solution of the above Geometric Brownian motion process is:

$$S_t = S_0 e^{\{(r - \sigma^2/2)t + \sigma W_t\}},$$

where the term S_0 is the initial stock price, the term W_t is a value of the standard Brownian motion process at time t .

We discretize the stock price process for simulation as follows:

$$S_{i,j} = S_{i,j-\Delta} e^{\{(r - \sigma^2/2)\Delta + \sigma \sqrt{\Delta} \varepsilon_{i,j}\}},$$

for $i = 1, \dots, m$ and $j = 1, \dots, n$, where m is the number of Monte-Carlo paths, n is the number of time steps used in the simulation. The discretization time step is assumed constant at Δ . The term $\varepsilon_{i,j}$ is the standard normal random variable sampled for simulation path i at time j .



In the ThetaML code examples, the cash flows in the option are denominated in currency 'CUR'. The parameters for the stock price 'S' are set as follows: the mean growth rate of the stock price is the same as the risk-free interest rate 'r' in currency 'CUR'; the volatility of the stock price is denoted as 'sigma', the initial stock price is 'S0'.

Since the cash flows occur at future times, a discount factor should be computed to obtain the present value of future cash flows. The discount factor in this example is a discount process with unit initial value in currency 'CUR'. We assume that interest rates are constant throughout the option life time and set the constant interest rate to 'r', such that the value of the discount process decays at a constant interest rate 'r'. When the future cash flows of an asset are multiplied by the discount process 'CUR', it serves two purposes: first, it discounts the future cash flows to current time; second, it translates the future cash flows into the currency unit 'CUR'.

The ThetaML model for simulating the stock price process and discount factor process is as follows:

```

1: model S_CUR_Processes
2: % This model simulates: stock prices 'S' that follow the
3: % Geometric Brownian motion process, and a discount process
4: % 'CUR' for constant interest rate r in currency 'CUR'
5:     import S0      "Initial stock price"
6:     import r       "Risk-free interest rate in currency CUR"
7:     import sigma   "Volatility of stock prices"
8:     export S       "Simulated GBM stock prices"
9:     export CUR     "Simulated discount process in currency CUR"
10:
11:     fork
12:         % initial stock price
13:         S = S0
14:         % infinite loop
15:         loop inf
16:             % time passing of @dt units, the value(s) of @dt are

```



```

17:         % determined later when the simulated stock price
18:         % process is applied in pricing financial contracts
19:         theta @dt
20:         % update stock prices for the time step @dt
21:         S = S * exp( (r - 0.5*Sigma^2) * @dt
22:         + sigma * sqrt(@dt) * randn() )
23:
24:     end
25: end
26: % initial values of the discount factor process; set to 1
27: % in currency 'CUR'
28: CUR = 1
29: % infinite loop
30: loop inf
31:     % time passing of @dt units, the value(s) of @dt are
32:     % determined later when the simulated discount factor
33:     % process is applied in pricing financial contracts
34:     theta @dt
35:     % the value of the discount factor decays at a constant
36:     % rate of r
37:     CUR = CUR * exp( -r * @dt )
38: end
39:
40: end

```

In the model `S_CUR_Processes`, line 5 - line 7 import respectively the initial stock price `s0`, the risk-free interest rate `r`, and the volatility of the stock price `sigma`, as constant parameters into the model.

The body of the model starts from line 11 and ends at line 38. The stock price process (line 13 - line 24) and the discount bond process (line 28 - line 38) are virtually paralleled processes. These two processes share the same model time axis. The use of the `fork ... end` statement at line 11 enables virtual parallelization of the two processes. It is as if the second code block from line 28 to line 38 is forked by the `fork ... end` statement as well.



In ThetaML, process variables implicitly incorporate scenario and time indexes. As such, the stock prices s and the discount bond CUR have both scenario and time dimensions. Henceforth, we talk about stock prices and discount bond prices in plural form.

The stock price s is initialized with s_0 at line 13. It evolves in time through the infinite loop `loop inf` that loops for all times and does value updates at a time interval `@dt`. The length of the infinite loop is determined later in the pricing model `American_put` listed below. The model `American_put` imports, among others, the stock price process s as parameter argument. In the model `American_put`, the imported stock price process s synchronizes with other imported process (such as CUR) and with the internal process V_{CUR} .

Inside the infinite loop `loop inf`, time is passed by the `theta` command at an interval `@dt`. When the model `S_CUR_Processes` is used in the pricing application `American_put`, the `@dt` parameter extracts the time interval, from current model time to the next nearest model time, passed by the `theta` command. In this case `@dt` evaluates to T/n in the model `American_put`.

In the model `S_CUR_Processes`, line 21 - line 22 updates the Geometric Brownian motion stock prices s for the time step `@dt`.

The discount factor CUR is initialized at 1 CUR at line 28. The value of CUR is then updated at a constant interest rate r for all times (by the infinite loop) with time step `@dt`. The length of the infinite loop `loop inf` and the value of the time step `@dt` are again determined in the model `American_put` where the processes s and CUR are imported as model arguments.

Having simulated the stock prices s and the discount process CUR , we next turn to the task of pricing an American put option. The ThetaML model for an American put option is as follows:



```

1: model American_put
2: % This model computes the price of a continuously
3: % exercisable put option based on 52 exercise dates. This
4: % model imports the stock price process 'S' and the discount
5: % factor process 'CUR', both are simulated in the model
6: % S_CUR_processes
7: import S      "Stock price process"
8: import CUR    "Discount process for the currency CUR"
9: import K      "Strike price for the American put option"
10: import T      "Option maturity time"
11: export P      "American put option values"
12:
13: % time 0 American put option values
14: P = V_CUR!
15: % n number of loops
16: n = 52
17: loop n
18:     % conditional evaluation of American put option
19:     % holding value compared with immediate exercise value,
20:     % both are discounted to time 0
21:     if E(V_CUR!) < (K - S) * CUR
22:         V_CUR = (K - S) * CUR
23:     end
24:     %time passing of 'T/n' units in ThetaML
25:     theta T/n
26: end
27: %American put option payoff at maturity time T,
28: %discounted by 'CUR' to time 0
29: V_CUR = max(K - S, 0) * CUR
30: end

```

In the example model `American_put`, line 7 and line 8 import respectively the stock price process `s` and the discount factor process `cur` that are simulated externally in the model `S_CUR_Processes`. Line 9 and line 10 import respectively the option strike price `K` and the option maturity time `T` as constant parameters into the model. Line 11 exports the time 0 American put option prices `P` distrib-



uted over all Monte-Carlo paths. Alternatively, We can export the option price P as a single estimate by simply setting $P = E(V_{CUR!})$ at line 14.

In ThetaML, process variables implicitly incorporate scenario and time indexes, the variables S , CUR and V in this model are processes with scenario and time indexes. Processes are time stepped by the `theta` command. When external processes such as S and CUR are imported into a ThetaML model, they synchronize with the processes inside the model, i.e. all the processes step forward in time along the same model time axis. Model time in the model `American_put` is spaced at a constant interval of T/n . Model time points are $j*(T/n)$, where $j = 1, \dots, n$ (n is the number of exercise times for the American put option). Model time points are such determined because at each time $j*(T/n)$, there is an event happening – we evaluate the possibility of early exercise. The processes S , CUR and V_{CUR} are synchronized at model time: if the variables S , CUR and V_{CUR} appear in the same code statement, their respective values are evaluated at the same model time executed at that line of code statement. If the code statement $V_{CUR} = (K - S)*CUR$ is evaluated at model time $j*(T/n)$, this is equivalent to assign to the variable V_{CUR} at model time $j*(T/n)$, the value of K minus the values of S at model time $j*(T/n)$, discounted to time 0 by CUR maturing at model time $j*(T/n)$, for all Monte-Carlo scenarios.

Since the ThetaML model `American_put` is programmed in chronological order, we go through the code statements according to the coding sequence.

At time 0 (line 14), the option prices P is set to be the same as the prices of the variable V_{CUR} . The variable V_{CUR} is referenced by a future operator “!”. The future operator “!” allows the values of V_{CUR} at a future time to be accessed. It looks into the future times to determine the current values of V_{CUR} .

The process of determining the future values of V_{CUR} is as follows: we enter a fixed loop of length n (line 17) with constant time interval T/n . The `if ... end`



block at line 21 - line 23 evaluates the early exercise decisions for the American put option; it compares pathwise the discounted holding values $E(V_CUR!)$ with the discounted immediate exercise values $(K - S)$ (both are discounted to time 0) and updates accordingly the values of V_CUR . This is equivalent to compare $E(V_CUR[i, j]!)$ with $(K - S[i, j]) * CUR[i, j]$, where, for illustration purposes, $[i, j]$ denotes the i - and j -th element of the matrices V_CUR , S and CUR . The subindex i denotes the i -th index element of the Monte-Carlo scenarios, the subindex j denotes the j -th index element of the discrete time grid, for $i = 1, \dots, m$ and $j = 1, \dots, n$, where m is the number of Monte-Carlo paths, n is the number of time steps. In case the if condition evaluates to be true for the index element $[i, j]$, the value of $V_CUR[i, j]$ is assigned $(K - S[i, j]) * CUR[i, j]$; this is done for all paths and time steps where the if condition is true. The whole evaluation and assignment process, across m Monte-Carlo paths and along n time steps, is compactly summarized by the following ThetaML code statements

```

21:  if E(V_CUR!) < (K - S) * CUR
22:      V_CUR = (K - S) * CUR
23:  end

```

The `theta` command (line 25) passes time T/n to the next model time point. After n loops with each loop passes by T/n time step, at the option maturity time T ($n * (T/n)$), the variable V_CUR is assigned the option payoffs $\max(K - S, 0) * CUR$ (line 29). That is, for all Monte-Carlo paths, the variable V_CUR at option maturity time T is assigned the discounted put option payoffs. The put payoffs are the maximums of 0 and the strike price K minus the stock prices s at maturity time T . Note that the maximums are respectively taken over all the Monte-Carlo paths. The process s and CUR are simulated up till time T in the model `American_put`. This is so because the infinite loop `loop inf` in the process simulation model `S_CUR_Processes` runs until the fixed length loop in the pricing model `American_put` stops running, which is time T . The time interval parameter `@dt` in the model `S_CUR_Processes` takes the constant value T/n . Since T/n is the constant



time interval passed by the `theta` command to the next model time point, for all model times in the model `American_put`.

The parameter `CUR` is the discount factor process simulated externally in the model `S_CUR_Processes`, it serves to discount a price process when the price process is multiplied by it. For example, the option payoffs at maturity time T is discounted with `CUR` by a factor equivalent to having a discount bond at time 0 with maturity T ; the values of `v_CUR` at time $j*(T/n)$ are normalized with `CUR` to time 0 by a discounting factor with maturity $j*(T/n)$. The same discount factor `CUR` that matures at each model time is applied to all Monte-Carlo paths, since in this example interest rate is assumed to be constant when externally simulating the process `CUR`.

Since the option payoffs are discounted by `CUR` (line 29), for consistent comparisons and compact coding, the statement $(K - S)$ in the `if` condition (line 21) is multiplied by `CUR` as well. Discounting to time 0 using `CUR` is equivalent to evaluating all future cash flows in the currency `CUR` in present value terms.

Computationally, we go backwards in time: starting from the American put option's final payoffs, on each exercise date, we evaluate, for all Monte-Carlo paths, the possibility of exercise and update the discounted option values accordingly. Thus iterate backwards to arrive at the current values for the put option.

The model `American_put` is programmed forward in model time and computationally evaluated backwards. This feature is realized with the future operator `!` and the `theta` command.

Since the time 0 values of `v_CUR` are referenced by a future operator `!` (line 14). We look into the future for the instance where the variable `v_CUR` is assigned some values. This occurs at the option maturity time T , when the variable `v_CUR` is assigned the discounted put option payoffs (line 29).



Denote $V_CUR[i, j]$ as the price of v_CUR at time j for the simulation path i , for $i = 1, \dots, m$ and $j = 1, \dots, n$, where m is the number of Monte-Carlo simulations, n is the number of time steps. At the option maturity time T , the variable v_CUR is denoted as $V_CUR[i, T]$ for simulation path i , where $T = n \cdot (T/n)$, and T/n is the discretization time interval. The immediate previous instance of v_CUR (i.e. $V_CUR[i, T - (T/n)]$) is at line 21 and is referenced with a future operator “!”. This future operator “!” accesses its next instances of v_CUR (i.e. $V_CUR[i, T]$) at line 29, so that at line 21 $V_CUR! = \max(K - S, 0) \cdot CUR$. At time $T - (T/n)$, the $E()$ function computes the expected value of $v_CUR!$ (equivalent to $V_CUR[i, T]$) conditional on the information known at time $T - (T/n)$.

The `if` condition ($E(V_CUR!) < (K - S) \cdot CUR$ at line 21) compares the expected discounted option holding value $E(V_CUR!)$ with the discounted option intrinsic value $(K - S) \cdot CUR$, respectively for all Monte-Carlo paths. If the condition evaluates to be true, the variable v_CUR at time $T - (T/n)$ is assigned the discounted option intrinsic value at time $T - (T/n)$, the assignment is done respectively for all the Monte-Carlo paths at time $T - (T/n)$. This process is computationally iterated back to time 0 to arrive at the time 0 values of v_CUR .





4 The ThetaML Type System

We shall see in this chapter how ThetaML helps

- To define the correct type of a process or variable and how
- Theta Suite can automatically create an appropriate GUI form for the data entry.



On the surface ThetaML looks like an untyped language. All variables can have different types without the need to explicitly declare their types. Internally however, ThetaML is strictly typed. Once the code is analyzed and its execution sequence is optimized in terms of memory and speed, all variable types are fixed. Although the ThetaML compiler implements sophisticated type extraction algorithms, sometimes it may fail to derive the desired variables types.

There are two reasons to help ThetaML with type extraction. First, the definition of imported variables allows customized widgets to show up in the ThetaML Configurator. Second, manual type definitions are needed for external models and functions. Because these models are defined outside the ThetaML language, their types cannot be derived automatically. The default return type for all external functions is a scalar value or a vector with one scalar value per Monte-Carlo path.



4.1 The Boolean Type

Boolean types are rendered as a check box in the ThetaML Configurator.

A variable is implicitly assumed **Boolean** if it is used in an **if** statement, as shown in the following code example:

```
model importBooleans
    import a,b,c,d "All booleans"

    if a & (b | c)
        %do something
    end

    if d
        %do something else
    end
end
```

A variable can be explicitly declared **Boolean** with the **type** keyword after the **import** statements.

```
model importsABoolean
    import B "Yes or No"

    type B Boolean

    %do something
end
```



4.2 The File Type

Files are internally treated like strings, but they behave differently in two ways. First, the ThetaML Configurator shows file selection buttons that allow the user to choose a file's location, rather than enter it as text. Second, file strings can contain paths that are relative to the **Theta Suite** workspace. Whenever a file location is passed to an external model or an external function, this file string is automatically converted to an absolute file path.

There are two types of files: input files and output files. For input files, a file selector is generated that only shows existing files. Output files allow the choice of using nonexistent files. The output file type can only be used within the context of external models, because ThetaML itself does not provide any language features for writing files to a file system. The following code example shows the use of file types in ThetaML.

```
model fileInAndOutWithCall
    import fileIn
    import fileOut
    %set file type in call
    %do something
end

model fileInAndOutExplicit
    import fileIn
    import fileOut

    type fileIn file
    type fileOut outputfile
    call @matlab : fileManipulatingModel
        export fileIn, fileOut

    %set file type explicitly for matlab function
    res = fileWriter(fileIn, fileOut)
    %do something
end
```



4.3 The Enum Type

Some models only have a limited number of input values. For these cases it is often desirable to generate a selection box in the ThetaML Configurator so that users of the model cannot enter incorrect values. They can only choose from a list of given values.

Enum types are implicitly created in the **Theta Orchestrator**¹¹ in Theta Suite. Inserting a ‘Switch’ with a list of named cases internally creates an `Enum` type with each case name as possible value.

Enum types can also be explicitly created with the `type` statement. The options are given as a list of possible values:

```
model someChoices
  import x "This variable can be 1,2,3, or 3.141"

  type x Enum [1, 2, 3, 3.141]

  %do something
end
```

¹¹ For details see Theta Suite Help > Theta Suite User Guide > Theta Suite Components > Theta Orchestrator



4.4 Array types

In ThetaML, it is essential that array lengths are known at compile time. Therefore the array length is part of the variable type, must be fully determined, and can not change during model executions.

Normally, array lengths are implicitly determined through their index accesses or definitions. These statements all specify the variable `x` to be an array of length 4.

```
a = 1 + x[4]
x[4] = 4
x = [1,2,3,4]
```

Array types can also be explicitly defined with the `type` statement. Arrays of numbers must be specified with `float` as a base type, followed by the array dimension in brackets.

```
model xHasLengthFour
```

```
  export x
```

```
  type x float[4]
```

```
  x = 3 %array with 4 elements, all set to 3
```

```
end
```





5 ThetaML Interfaces

ThetaML offers a very compact notation for sophisticated financial models. However, as for all programming languages, inappropriate use of some ThetaML language syntax can be misleading.

ThetaML provides interfaces to ensure that certain functionalities are within the constraints of rational financial modeling. Interfaces can be applied to a ThetaML model. If the model does not comply with the constraints defined by the interface, the model can not be executed and returns an error message with the violated constraint. Thus, model reviews can rely on the presence of certain interfaces, without the need to check each sub_model for inappropriate use of ThetaML language features.



5.1 Interface Syntax

The ThetaML interface is defined similar to a `model`, starting with the `interface` keyword. The interface contains sections for obligatory variables and constraints for inputs and language features.

The ThetaML `interface` has the following structure:

```
interface exampleFace
%impose some constraints
end
```

A model that complies with the interface must indicate compliance with the `implements` keyword. The compliant model is sometimes also called implementing model.

```
model compliantModel implements exampleFace
%model content
end
```



A single interface can be applied to multiple implementing models sharing the same import argument(s):

```
% the interface Itest imposes a constraint on S0 which
% is imported in the implementing models stock1 and stock2

interface Itest
    import S0 "Initial stock price"

    % impose constraint on the initial stock price
    assert(S0 > 0, 'Initial stock price must be greater than 0')
end

model stock1 implements ITest
    import S0      "Initial stock price"
    import mu1     "Drift of stock price"
    import sigma1  "Volatility of stock price"

    % model content
end

model stock2 implements ITest
    import S0      "Initial stock price"
    import mu2     "Drift of stock price"
    import sigma2  "Volatility of stock price"

    % model content
end
```



Interfaces can also be extended with additional constraints by applying an **interface** to an **interface**. Again the **implements** keyword is used:

```
interface IStricter implements ITest
% impose additional constraints upon the existing
% interface ITest
```

```
end
```

The following example illustrates how to use **interface** in ThetaML:

Example 36:

```
1: interface ITestInterface
2: % This interface applies to the implementing model GBM
3:   import r "Risk-free interest rate"
4:   %impose a constraint on interest rate r
5:   assert(r > 0, 'Interest Rate r must be greater than 0')
6:
7: end

1: model GBM implements ITestInterface
2: % This model simulates stock prices following a Geometric
3: % Brownian Motion process
4:   import S0          "Initial stock price"
5:   import r            "Risk-free interest rate"
6:   import sigma        "Volatility of stock price"
7:   export St           "Simulated stock Prices"
8:
9:   St = S0             % initial stock price at S0
10:  dt = 1/12            % discretization time interval
11:
12:  loop 12              % the loop runs 12 times
13:    theta dt           % time passing of dt = 1/12
14:    % update the GBM stock price for the time step dt
15:    St = St * exp( (r - 0.5 * sigma^2) * dt
```



```

16:             + sigma * sqrt(dt) * randn() )
17:         end
18:     end
19: end

```

The **interface** `ITestInterface` imposes a constraint on the interest rate parameter `r` used in the model `GBM`. The constraint is imposed using the ThetaML keyword `assert` (line 5). The `assert` statement takes as first argument the condition that must be satisfied by the respective parameter(s), followed by a comma, then a string¹² that shows up as error message¹³ in the ThetaML Configurator when the constraints are violated.

The model `GBM` indicates compliance with the **interface** `ITestInterface` using the keyword **implements**. `GBM` is a ThetaML model for simulating the discretized Geometric Brownian Motion (GBM) process of stock prices. The stock price process `st` starts at `s0` (line 9), the discretization time interval is a constant `dt` equal to `1/12` (line 10). The command `theta dt` (line 13) passes time to the next discretization time point. The `st` process is updated 12 times at a constant time interval `1/12`. Line 15 - 16 call three math functions `exp`, `sqrt` and `randn()`. The resulting simulated stock prices are exported at line 7. Note that the variable `st` is a process variable with implicit scenario index and time index.

¹² String is a ThetaML internal type.

¹³ The error message can be found in the text box under the "Info" section in the ThetaML Configurator.

5.2 Interface Import and Export Statements

ThetaML interfaces may contain **import** and **export** statements. Variables listed there must be imported and exported by every implementing **model**. The **interface** can thereby guarantee the presence of certain variables and accidental re-naming can be avoided. Furthermore, variables in the **interface** can be provided with a default comment for the implementing **model**, as the following example shows:

```

interface ImustExport
    % the exported variable S also adds a default comment for the
    % variable S in the implementing model 'modelS'
    export S "The stock price of XXX"

    % impose constraints for the exported variable S, note the
    % constraints on S are checked after the model modelS is run

end

model modelS implements ImustExport
    % import statements ...
    export S

    % process statements

end

```



5.3 Language Constraints

Language constraints can be applied to models, including all sub_models. If a **model** with language constraint(s) calls another model which uses that language feature but does not itself implement that constraint, an error marker is shown for the calling operation. No error is shown for the called model.

The following language constraints can be applied to ThetaML. All constraints are case insensitive.

@notheta: This model can not pass model time using the **theta** command. All computations must be done based on the present value of state variables. It can not have any process dependency.

@noBackward: No use of the future operator “!”. Access to the future value(s) of a variable and reverse execution order is forbidden.

@noRegression: No use of the expected values $E()$ and the betas $Beta()$. Because these operators have an implicit access to the future values of variables, this constraint is automatically active if **@noBackward** is included in the constraints.

@noExternalFunction: Calls to external functions, such as those defined in Matlab, are forbidden. All functionalities must be defined in ThetaML. This constraint ensures portability of the ThetaML model to other numerical backends.

@noExternalModule: Calls to external models are forbidden. This constraint is automatically active, if **@noExternalFunction** is active. Models are invoked with the **call @backend:ModelName**. Unlike functions, models can also **import** and **export** time dependent processes.



@noInfLoop: The use of `loop inf` is restricted. The infinite loop construct allows the creation of stochastic processes that are simulated as long as needed. The use of this construct is inappropriate in product- or contract-specific models.

@noFork: Forbids the `fork` keyword. It is important to point out that calls to `sub_model` implicitly create a `fork`. i.e., the `sub_model` bodies are executed in parallel (in model time) with the calling model. Calls clearly separate variable spaces and are hence easier to handle than forks. Forks are helpful for small and experimental models, but should not be used in complex projects.

An example for an `interface` activating all available constraints can look like this:

```
interface IConstrainAll
    @notheta
    @noExternalFunction
    @noBackward
    @noExternalModule
    @noFork
    @noInfLoop
end
```



5.4 Value Assertions

Another type of constraint can apply to input parameters. These constraints do not concern the ThetaML model, but the input parameters defined in the ThetaML Configurator. If any of these assertions are violated, the errors are shown in the configuration, not in the model.

Value assertions are specified in the **interface** with the `assert` statement. An assertion has two arguments. The first is a boolean expression that evaluates whether the constraint is valid. The second optional argument is a human readable error message that is shown in the configuration page if the assertion fails.

An example for using the `assert` statement in a ThetaML **interface** looks like this:

```
interface IAssertRanges
    import sigma, rho
    % use 'assert' to impose the constraints for the input
    % parameters

    % the input parameter 'sigma' must be bigger than 0
    assert(sigma > 0, 'Sigma must be positive')

    % the input parameter 'rho' must lie between -1 and 1
    assert(rho >= -1 && rho <= 1, 'Correlation must be between
        -1 and 1')

end
```





6 Workflows

Workflows in ThetaML are very similar to the models we saw in the previous chapters. They allow automation of recurring tasks, especially the definition of data pre- and post-processing. Typical applications of workflows are in defining iterations over the starting values of a given pricing task. Workflows can also define sources from which input data are read or result files that should be written to. Thus, workflows provide the capabilities one would expect from a normal scripting language, without support for ThetaML simulation specific features such as computing expected values (the \mathbb{E} function), future access using the future operator “!”, or scenario-wise executions.



Workflows look very similar to ThetaML simulation models¹⁴. However, they are evaluated differently from the compilation and optimization method used in evaluating ThetaML simulation models. Workflows are evaluated step-by-step according to the order of program flow, and they allow much more flexibility when it comes to complex data types. Moreover, workflows can take full advantage of the infrastructure integrated into Theta Suite. This includes automatic generation of graphical user interfaces, automatic extraction of data types, and editing support with instantaneous error indicators.

¹⁴ Simulation models here refer generally to models defined using the keyword model.

6.1 Workflow Definitions

Workflows are introduced with the keyword **workflow** followed by the workflow name:

```
workflow <WorkflowName>  
...  
end
```

Workflows can **import** and **export** variables similar to ThetaML simulation models:

```
workflow testFlow  
    import X "comment for X"  
    import Y "comment for Y"  
    export report "Result object contains all interesting stuff"  
    ...  
end
```



Workflow variables can have the input data types listed in Table 3. Wherever possible, these types are automatically extracted from the workflow source with the usual type extraction method¹⁵ in ThetaML. Workflows are dynamically typed. Type definitions are only relevant for the user interface, where custom widgets are shown.

Data Type	Widgets
Boolean	Check box
Enum	Selection box
File	File Chooser for existing files
Outputfile	File Chooser for existent and non-existent files
Float/String/Array	Text field for free form formulas

Table 3. Input data types defined in workflows.
Column 1 lists the input data types in workflows. Column 2 gives a short description of the data types in the form of widgets that appear in ThetaML Configurator.

¹⁵ For more on data extractions in ThetaML, please see section 1 of Chapter 3.

6.2 Workflow Statements

Workflows allow the same set of statements as ThetaML simulation models, except that workflows do not support the `theta` and `fork` statements. Workflows do not maintain their own model time. The statements in workflows are always executed in the order they are defined.



6.3 Assignments in Workflows

Assignments are definitions of variables. They can operate on variables, their subfields, and array components. The right hand side of an assignment can be any value or formula:

```
% assign 'X' some values
X = ...

% assign something to 'field2' of the object 'X';
% 'field2' is a subfield of 'field1'; 'field1' is a field
% of the object 'X'
X.field1.field2 = ...

% assign something to the 'index' element of the array 'field2';
% 'field2' is a subfield of the 3rd index element of the
% array 'field1'; 'field1' is a field of the object 'X'
X.field1[3].field2[index] = ...
```



6.4 Loops in Workflows

Loops share their semantics with ThetaML simulation models. They can be defined by following the `loop` keyword with either a fixed number or an array with values. Compared to ThetaML simulation models, infinite loops (`loop inf`) are forbidden.

A fixed loop example:

```
loop 10
    % do this ten times
end
```

Array loops with `indexOf()` are possible:

```
% x serves as an iterator for the 'Array'
loop x : Array
    % loop through each element in 'Array'

    % optional using indexOf()
    index = indexOf(x)
end
```

Loop results can be put into an object and exported:

```
% export the 'report' object
export report
% define the 'ShiftArray'
ShiftArray = [-0.5, -0.1, 0, 0.1, 0.5]
% loop through the elements of the array 'ShiftArray'
% 'shift' serves as an iterator for the array 'ShiftArray'
loop shift : ShiftArray
    % during the looping process, store in the field 'shifted'
    % of the object 'report' the assigned values;
    % the compiler automatically detects 'report'
    % is of type object
    report.shifted[indexof(shift)] = ...
end
```



6.5 Conditional Executions in Workflows

Workflows use the same conditional evaluation statements as ThetaML simulation models:

```
if <condition>  
    % do if the 'condition' is true  
else  
    % do if the 'condition' is false  
end
```



6.6 Sub Workflows

Workflows can call each other using the syntax for calling sub_models in simulation models:

```
%call sub workflow 'subworkflow'
call subworkflow
    % export the 'param' field of the object 'context' defined
    % in the calling workflow to the 'param' argument in the
    % called 'subworkflow'
    export context.param to param
    % import in the 'case_1' field of the object 'report' defined
    % in the calling workflow from the 'report' argument in the
    % called 'subworkflow'
    import report.case_1 from report
```



6.7 Functions

Workflows support all functions defined in ThetaML simulation models, except the expected value function (\mathbb{E}), the beta function (Beta) and the future operator (“!”).

Below we give a list of some supported operators and functions in workflows:

- Arithmetic operators: +, -, /, *, ^
- Logical operators: ~, &&, ||
- Array operators: x[5], [1, 2, 3], [1:2:50], [1:10], `length()`, ...
- Mathematical functions: `max`, `min`, `log`, `exp`, `sqrt`, `sin`, `cos`, `atan`, ...

Additionally workflows have access to file operations. The following are two workflow commands that operate on files:

`load`: this command load a .thetaml file, such as a pricing configuration file

`run`: this command runs the pricing configuration file



The following is a simple workflow example:

```
workflow simpleWorkflow
    import testFile    "Configuration file: testFile"
    import shift        "Shift level for initial parameters"
    export result       "Export the results"

    % load the ThetaML configuration file 'testFile'
    conf = load(testFile)

    % change the parameter value by 'shift';
    % 'param' is a field defined in the variable object 'conf'
    % 'value' is a subfield of the field 'param'
    conf.param.value = conf.param.value + shift

    %return the results ran in the object 'result'
    result = run(conf)
end
```



6.8 External Namespaces

Workflows can make use of all the functionalities that are available from the ThetaML Configurator. That includes the ThetaML `@today` parameter, functions following the `@matlab` call command, and the `@ql` QuantLib namespace. If some customized extensions are added to Theta Suite, they are also accessible by workflows.

The compiler does not assume that results in workflows from the external functions are numerical arrays or associated with scenario indices. All return types (structs, arrays, objects, and so on) are therefore allowed in workflows.

Obviously Matlab actions can be triggered, such as storing data to disk, converting to Excel, or opening dialogs. Matlab statements that make modifications to the Matlab workspace can have unexpected impact on the running workflows. They may work, but are not universally supported.





7 ThetaML Language by Example

The two tutorials in this chapter serve as a guide to implement your first financial model:

- The Tutorial “From European to American” shows how to model classical equity options, starting from a simple European option up to a more complex Compound option with early exercise features.
- The Tutorial “Hedging in ThetaML” presents different ways for option hedging. The hedging strategies go from simple delta hedging to more efficient hedging with our `Beta` function.



7.1 Tutorial From European to American

This tutorial applies the ThetaML Language in the area of pricing financial derivatives.

ThetaML presents the definition of complex financial derivatives with unprecedented simplicity. This is demonstrated by the following examples.

To start with, we implement in ThetaML a simple European option, then show how easy it is to add additional features – such as Bermudan and American exercise features, or to change the underlying to another option, such as a compound option.

7.1.1 The Stochastic Process

To price an option, we first define a stochastic process for the underlying. As there might be thousands of scenarios for the future values of the underlying, good modeling language and powerful simulation engine are essential for estimating accurate future values. This is where ThetaML comes into play.

For the simple purpose of this tutorial, we use the stock price process s and the discount factor process cur simulated in the model `S_CUR_Processes` in the Chapter Example of Chapter 3. The stock price parameter s follows a Geometric Brownian motion process under the risk-neutral measure, the discount factor process cur is discounted at a constant interest rate r .

In terms of naming conventions in ThetaML models, we use in general the suffix `_CUR` for variables that are discounted to time 0 by the discount factors cur . For example, in the code statement `v_CUR = max(K - S, 0) * cur`, the variable `v_CUR` has suffix `_CUR`, because its right hand side values are discounted by cur to time 0. By discounting to time 0, we always talk about future cash flows in present value terms.



With the simulated `s` and `CUR` processes, we now turn to the task of pricing a European option in ThetaML.

7.1.2 European Option

This little example shows how to price a European option in ThetaML.

```

1: model EuropeanPut
2: % This model returns a simulated European put option price
3:   import S      "Stock prices"
4:   import CUR    "Discount factors"
5:   import K      "Strike price for the European put option"
6:   import T      "Time to maturity in years"
7:   export P      "European put option price"
8:
9:   % time 0 European put option price,  $E(V_{CUR!}) = E(V_{CUR})$ 
10:  P = E(V_CUR!)
11:  % T years pass
12:  theta T
13:  % at maturity T, the option payoff is discounted to time 0
14:  V_CUR = max(K - S, 0) * CUR
15:
16: end

```

In the model `EuropeanPut`, the `import` block (line 3 - 6) defines the variables that must be imported as model arguments, among which, the stock prices `s` and discount factors `CUR` are externally simulated processes. When external processes are imported into the model, they automatically synchronize in model time with the processes in current model. As such, `s` and `CUR` take their respective values at the corresponding model time. For example, `s` and `CUR` at line 14 take their respective values at maturity time `T`.

The exported variable `P` (line 7) is the option price returned by the model.

Following the `import` and `export` block, the body of the model consists of three statements.

At time 0, the option price P is defined as the expected value of the discounted future payoffs v_{CUR} conditional on the time 0 information (line 10). The variable v_{CUR} at line 10 is referenced with the future operator “!”. The future operator “!” acts like a function on v_{CUR} , which means we wait to determine its values till a later instance when v_{CUR} is explicitly assigned some values.

At line 12, the `theta` command advances model time by T years.

Then we reach maturity time T , and the variable v_{CUR} is assigned the discounted option payoffs (line 14). The option payoffs v_{CUR} at line 14 are discounted to time 0 by the discount factors CUR that mature at time T , as such future cash flows are represented in present value terms.

The variables s , CUR and v_{CUR} are process variables with implicit scenario and time indexes, as such we always talk about their values in plural forms.



7.1.3 Bermudean Option

We now extend the European option with an additional feature. Suppose that half the maturity time has passed and we decide to exercise the option early. Let us adjust our model to reflect this new feature.

```

1: model BermudanPut
2: % This model returns a simulated Bermudan put option price
3: import S    "Stock prices"
4: import CUR   "Discount factors"
5: import K     "Strike price for Bermudan put option"
6: import T     "Time to maturity in years"
7: export P     "Bermudan put option price"
8:
9: % time 0 Bermudan put option price
10: P = E(V_CUR!)
11: % T/2 years pass
12: theta T/2
13: % early exercise evaluation, compare expected discounted
14: % option hold value with discounted option intrinsic value
15: if E(V_CUR!) < (K - S)* CUR
16:     V_CUR = (K - S)* CUR
17: end
18: % another T/2 years pass
19: theta T/2
20: % at maturity T, V_CUR has the discounted put payoffs
21: V_CUR = max(K - S, 0)* CUR
22:
23: end

```

The model `BermudanPut` imports two external processes: the stock prices `s` and the discount factors `cur`. The two processes `s` and `cur` automatically synchronize in model time with other processes in current model. As such, `s` and `cur` take their respective values at the corresponding model time. For example, `s` and `cur` at line 15 take their respective values at time $T/2$. At line 21 they take their respective values at option maturity time T .



In the model `BermudanPut`, initially we define the option price P as the expected discounted future values of v_{CUR} (line 10), and wait for a time period of $T/2$ (line 12). We then compare the expected discounted option hold values $E(v_{CUR})$ with the discounted option intrinsic values $(K - S) * CUR$ (line 15). If the `if` condition at line 15 turns out to be true, we assign the new values to v_{CUR} (line 16). We then wait another $T/2$ years (`theta T/2` at line 19) and assign the discounted payoff values to v_{CUR} at the option's maturity time T (line 21). The variables S , CUR and v_{CUR} are process variables with implicit scenario and time indexes, as such we write their values in plural form.

Note that the compiled codes will record the commands into a computational order. This forces the final assignment of v_{CUR} to be evaluated first. The result is then overwritten by optimized option values when stepping backwards in time.



7.1.4 American Option

An American option goes even further than the Bermudan option in that it can be exercised continuously. To implement this in ThetaML, we formulate a Bermudan approximate with finite step size between possible exercise times. The following ThetaML model implements such an approximation with daily exercise intervals, assuming 252 trading days per year.

```

1: model AmericanPut
2: % This model returns a simulated American put option price
3:   import S      "Stock prices"
4:   import CUR    "Discount factors"
5:   import K      "Strike price for American put option"
6:   import T      "Time to maturity in years"
7:   export P      "American put option price"
8:
9:   % time 0 American put option price
10:  P = E(V_CUR!)
11:  % loop T*252 times, T*252 rounded down to
12:  % the nearest integer
13:  loop T*252
14:    % early exercise evaluation
15:    if E(V_CUR!) < (K - S)* CUR
16:      V_CUR = (K - S)* CUR
17:    end
18:    % 1 trading day passes
19:    theta 1/252
20:  end
21:  % at time T, the option payoff is discounted to time 0
22:  V_CUR = max(K - S, 0)* CUR
23:
24: end

```

In the model `AmericanPut`, at time 0, we define the option price `P` as the expected discounted future values of `V_CUR!` (line 10). With time passing (`theta 1/252`), on each exercise date, we compare the expected discounted values of continu-

ation $E(V_{CUR})$ with the discounted option intrinsic values $(K - S) * CUR$ (line 15). If the latter turns out favorably, we assign the new values to V_{CUR} (line 16). After $T * 252$ days, we assign the discounted payoff values to V_{CUR} at the option's maturity time (line 22). We discount the process variable V_{CUR} and the option intrinsic values $(K - S)$ to time 0 so that future cash flows are always seen in present value terms.

The model `AmericanPut` imports two external processes: the stock prices s and the discount factors CUR . The two processes s and CUR automatically synchronize in model time with processes in current model. As such, s and CUR take their respective values at the corresponding model times. For example, s and CUR at line 15 take their respective values at the model time executed at that line of code. At line 22 they take their respective values at option maturity time T . The variables s , CUR and V_{CUR} are process variables with implicit scenario and time indexes, their values are therefore written in plural form.

Computationally, we go backwards in time: starting from the American put option's final payoffs, on each exercise date, we evaluate the possibility of early exercise and update the discounted option values accordingly. Thus iterating backwards to arrive the current put option price.



7.1.5 Compound Option

The compound option is an option on an option. Let us assume the outer option is a European call option and the inner option is an American put option, i.e. a European call written on an American put.

Based on the above ThetaML models for European and American options, our model for the compound option is implemented as follows:

```

1: model CompoundOption
2: %This model simulate prices a Compound option
3:   import S    "Stock prices"
4:   import CUR  "Discount factors"
5:   import K1   "Strike price for the outer option"
6:   import K2   "Strike price for the inner option"
7:   import T1   "Time to maturity of the outer option"
8:   import T2   "Time to maturity of the inner option"
9:   export P    "Compound option price"
10:
11: % time 0 compound option price
12: P = E(V_CUR!)
13: % T1 time passes
14: theta T1
15: % outer European call option payoffs
16: if E(V_CUR!)- K1 * CUR > 0
17:   % if the payoffs have positive values, V_CUR at time
18:   % T1 is assigned (V_CUR! - K1*CUR); at time T1,
19:   % the values of V_CUR! remain to be determined
20:   V_CUR = V_CUR! - K1 * CUR
21: else
22:   V_CUR = 0
23: end
24:
25: % inner American put option
26: loop (T2-T1)*252
27:   %early exercise evaluation

```



```

28:         if E(V_CUR!) < (K2 - S) * CUR
29:             V_CUR = (K2 - S) * CUR
30:         end
31:         % time passing of 1 trading day
32:         theta 1/252
33:     end
34:     % at time T2, inner option payoff discounted to time 0
35:     V_CUR = max(K2 - S, 0) * CUR
36:
37: end

```

In the model `CompoundOption`, we **import** two strike prices K_1 and K_2 as well as two maturity times T_1 and T_2 respectively for the outer and inner options. We **export** the computed compound option price in the variable P .

The model `CompoundOption` imports also two external processes: the stock price s and the discount factors CUR . The two processes s and CUR automatically synchronize in model time with processes in current model. As such, s and CUR take their respective values at the corresponding model time. For example, s and CUR at line 28 take their respective values at the model time executed at that line of code. At line 35 they take their respective values at time T_2 .

Initially, the compound option price P is set to the expected discounted future values of $V_CUR!$ (line 12). The future values of V_CUR are determined as what follows.

At time T_1 , we reach the maturity of the outer European call option. The outer European call payoffs depend on the values of the inner American put option at T_1 . Instead of using the single line of code $V_CUR = \max(E(V_CUR!) - K_1 * CUR, 0)$ for the outer European call payoffs, we code the payoffs as in line 16 - line 23, which is a separate statement version for $V_CUR = \max(E(V_CUR!) - K_1 * CUR, 0)$ but numerically more efficient.

Assuming 252 possible exercise dates (line 26) for the inner American put, on each exercise date starting from T_1 , we compare the expected discounted Ameri-



can put holding values with its discounted intrinsic values and assign the updated values to v_CUR (line 28 - line 30). After $(T2-T1)*252$ days, we reach the maturity of the inner American put, and assign the discounted American put payoffs to v_CUR (line 35).

The discount factors CUR discounts the process variable v_CUR and the statement $(K2 - S)$ to time 0. By discounting to time 0, we always talk about future cash flows in present value terms. The variables S , CUR and v_CUR are process variables with implicit scenario and time indexes.

The compiler records the commands into a computational order, i.e. iterate backwards in time. Starting from the inner American put option's final payoffs, on each exercise date, we evaluate the possibility of exercise and update the discounted option values accordingly. Thus iterating backwards, at the outer European call option maturity $T1$, we already know the estimated inner American put option values at $T1$; as such, the outer European call option is easily valued like a vanilla European call.



7.1.6 Hedged American Option

With the above setting as our background, it is easy to introduce variance optimal hedge. The ThetaML command `Beta(S,V)` computes the optimal fraction of s that minimizes the variance of a portfolio with one option v and `Beta(S,V)` number of asset with price s .

Our model for the hedged American option is implemented as follows:

```

1: model Hedged_American
2: % This model beta hedges the American put option
3: import S           "Stock prices"
4: import CUR          "Discount factors"
5: import K            "Strike price for American put option"
6: import T            "Time to maturity in years"
7: export P            "Hedged American put option price"
8: export V_CUR        "Process of option hedge"
9:
10: % time 0 American put option price
11: P = E(V_CUR!)
12: % loop T*252 times, assuming 252 trading days in a year
13: loop T*252
14:     % using the Beta function to provide a better hedge
15:     % for the price process V_CUR, note
16:     % Beta(S!*CUR!, V_CUR!) = Beta(S*CUR, V_CUR)
17:     V_CUR = V_CUR! - Beta(S!*CUR!,V_CUR!)*(S!*CUR! - S*CUR)
18:     %time passing of 1 trading day
19:     theta 1/252
20:     %early exercise evaluation
21:     if E(V_CUR!) < (K - S)* CUR
22:         V_CUR = (K - S)* CUR
23:     end
24:
25: end
26: % the time T American put payoffs are discounted to time 0
27: V_CUR = max(K - S, 0)* CUR
28:
29: end

```



In the model `Hedged_American`, we import the stock prices `s` and the discount factors `cur` simulated externally. The two processes `s` and `cur` automatically synchronize in model time with processes in current model. As such, `s` and `cur` take their respective values at the corresponding model time. For example, `s` and `cur` at line 21 take their respective values at the model time executed at that line of code. At line 27 they take their respective values at option maturity time T .

At time 0 we assign to `P` the expected discounted future values of the hedged American put option `v_cur!` (line 11). The hedged option process `v_cur!` minimizes the variance of the hedge by investing `Beta(s!*cur!, v_cur!)` amount in the stock `s` (line 17). The function `Beta(s!*cur!, v_cur!)` computes the beta factor for its two arguments, where `s!*cur!` is the explanatory variable and `v_cur!` is the dependent variable. Assuming we are now at time t , the `Beta(s!*cur!, v_cur!)` function estimates the relationship between `s!*cur!` and `v_cur!` conditional on the information at time t , where `s!`, `cur!` and `v_cur!` take respectively the values of `s`, `cur` and `v_cur` at the immediate next model time. On each exercise date, we compare the expected discounted holding values of the American put option with its discounted intrinsic values and update accordingly the discounted option values (line 21 - line 23). At the option maturity time T , `v_cur!` is assigned the discounted American put payoffs (line 27). We discount to time 0 the processes `v_cur` and `s` as well as the expression $(K - s)$ so that future cash flows are always seen in present value terms.

The variables `s`, `cur` and `v_cur` are process variables with implicit scenario and time indexes, and their values are in plural form.

This completes the introductory tutorial “From European to American”.

7.2 Tutorial Hedging in ThetaML

7.2.1 Introduction

This tutorial introduces a variety of hedging techniques in ThetaML. As a byproduct, it also illustrates the simplicity and flexibility of using ThetaML for fast and accurate simulations.

To start with, we compute, in ThetaML, the very familiar Black-Scholes delta, then use this Black-Scholes delta to simulate hedge European put option - stepping both forward and backward in time. In the end, we compare the hedged portfolio value with the exact European put price. Our first results produce the same European put price for forward and backward hedges, with a standard error of 0.84 in both cases.

Next, we present our unique `Beta` function and illustrate by examples its wide applicability and flexibility in more advanced hedge settings. We start with the more familiar variance minimal hedge - for single and multiple underlyings and for single- and multi-dimensional stochastic processes. We then proceed to apply our `Beta` function to static and dynamic portfolio hedging. Finally, our `Beta` function is applied in the real world hedging with transaction costs, demonstrated by code examples for portfolio rebalancing at constant intervals and for position changes above a certain barrier level.

In terms of naming conventions in the example ThetaML models, we use the suffix `_CUR` for variables that are discounted to time 0 using the discount factors `CUR`. For example, in the code statement `v_CUR = max(K - S, 0) * CUR`, the variable `v_CUR` has suffix `_CUR`, because its right hand side values are discounted by `CUR` to time 0. By discounting to time 0, we always talk about future cash flows in present value terms.



To proceed with presenting various hedging methods, we set up a hedge portfolio Π as

$$\Pi = V + d * S$$

with an option V and a position of d in the hedge instrument S . The task of the hedging procedure is to choose d such that the portfolio Π does not change much even if the underlying S changes.

7.2.2 Delta Hedging

We start with a simple example where the process S is driven by a Geometric Brownian motion process with a drift equal to the risk-free rate r and volatility σ . This allows us to use the Black-Scholes-Merton equation to obtain a representation for d . The choice and use of d to hedge portfolio risk is called delta hedging.

The delta-hedge pricing method offsets the portfolio risk in such that it can be made completely risk free in the basic Black-Scholes economy. This is unrealistic in the real world, mainly because the hedge must be rebalanced continuously in time whereas a real trader can only buy and sell at discrete times. In this section, we will compute and analyze the errors that occur when using discrete instead of continuous hedges.

We use European put option as an example. The formula for European put option delta is known as

$$d = \frac{\partial V}{\partial S} = N\left(\frac{\log\left(\frac{S}{K}\right) + \left(r + \frac{\sigma^2}{2}\right)T}{\sigma\sqrt{T}}\right) - 1$$

where $N(\cdot)$ is the cumulative standard normal distribution, and the term $\frac{\partial V}{\partial S}$ refers to the partial derivative of V with respect to S .



With this formula and the convenience that ThetaML can call any regular Matlab functions, it is very easy to obtain the delta of a European put option in ThetaML as what follows.

For consistency, all subsequent code examples use the following assumptions and parameter values:

- Stochastic model for the stock price: Geometric Brownian Motion
- Volatility of the stock price : 0.4 (40% p.a.)
- Risk-free interest rate : 0.05 (5% p.a.)
- Current stock price : 100
- Initial discount factor value CU_R : 1
- Strike price of the option : 100
- Maturity time : 1



The following code example is a ThetaML model for computing Black-Scholes delta.

```

1: model BlackScholesDelta
2: % This model returns the Black-Scholes hedge delta for a
3: % European put option
4: import S           "Current stock price"
5: import K           "Strike price for the European put option"
6: import sigma       "Volatility of the underlying stock"
7: import r           "Risk-free interest rate"
8: import T           "Time to maturity"
9: export delta       "Black-Scholes delta"
10:
11: if T <= 0
12:     if S < K
13:         delta = 1;
14:     else
15:         delta = 0;
16:     end
17: else
18:     d1 = (log(S/K) + (r+(sigma^2)/2)*T) / (sigma*sqrt(T));
19:     delta = normcdf(d1,0,1) - 1;
20: end
21:
22: end

```

In the model `BlackScholesDelta`, line 18 uses two mathematical functions `log` and `sqrt`. Line 19 calls a Matlab function `normcdf` to compute values of the cumulative standard normal distribution.

Having computed the Black-Scholes delta in ThetaML, we proceed to use this analytical delta to simulate hedge portfolio values and compare the hedge error between the hedging portfolio and the option value. This is illustrated by the following code examples and hedge error histograms.

The following code example is a ThetaML model for simulating hedging portfolio values forward in time, using the Black-Scholes delta as the hedge strategy for the underlying hedge instrument. It also computes the hedging error based on the analytical Black-Scholes option delta.

```

1: model DeltaHedge_Forwards
2: % This model simulates forward in time the delta hedge of a
3: % European put option
4:   import S           "Stock prices"
5:   import CUR          "Discount factors"
6:   import sigma        "Volatility of the stock price"
7:   import r             "Risk-free interest rate"
8:   import K             "Strike price for the European put option"
9:   export Pi_CUR        "Hedging portfolio value"
10:  export delta         "Black-Scholes hedge delta"
11:  export error         "Hedging error"
12:
13:  T = 1 % time to maturity of the option
14:  n = 252*T % loop length
15:  % Pi_CUR has the same expected value as V_CUR at time 0
16:  Pi_CUR = E(V_CUR!)
17:
18:  loop n
19:    % obtain the Black Scholes delta
20:    call BlackScholesDelta
21:    export K, sigma, r, S
22:    % @time has passed, the maturity is now T - @time
23:    export T - @time to T
24:    import delta
25:    % update the previous stock price S_old
26:    S_old = S
27:    % update the previous discount factor CUR_old
28:    CUR_old = CUR
29:    % T/n time interval passes
30:    theta T/n
31:    % update the hedging portfolio by delta amount of S
32:    Pi_CUR = Pi_CUR + delta * (S*CUR - S_old*CUR_old)

```



```

33:     end
34:     % option payoffs discounted to time 0
35:     V_CUR = max(K - S, 0) * CUR
36:     % hedging error
37:     error = Pi_CUR - V_CUR
38:
39: end

```

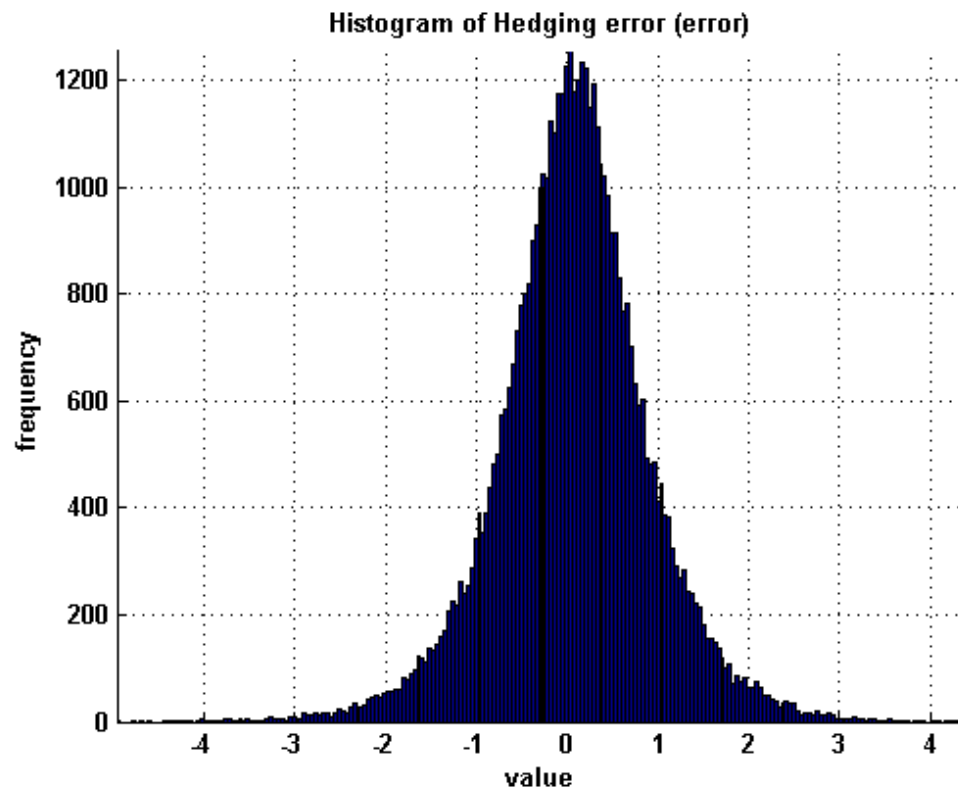
In the model `DeltaHedge_Forwards`, the European put option maturity T is set to 1 year (line 13), each year is assumed to have 252 trading days (line 14).

The model `DeltaHedge_Forwards` imports the stock prices s and the discount factors CUR simulated externally in the model `S_CUR_Processes` in the Chapter Example of Chapter 3. The two processes s and CUR automatically synchronize in model time with processes in current model. As such, s and CUR take their respective values at the corresponding model time. For example, s and CUR at line 26 and line 28 take their respective values at the model time executed respectively at that line of code. At line 35 they take their respective values at option maturity time T .

We initially set up the hedge portfolio Pi_CUR to have the same expected discounted value as the European put option V_CUR (line 16). With time passing at constant T/n intervals (`theta T/n` at line 30), we optimally hedge the portfolio Pi_CUR using the Black-Scholes `delta` as the position taken in the underlying stock s (line 32). This strategy is updated each time step T/n . At the option's maturity T (line 35), V_CUR is assigned the discounted option payoffs. Line 37 computes the hedging errors between the hedging portfolio values and the option values at option maturity T .

We discount to time 0 the processes V_CUR and s so that future cash flows are always seen in present value terms. The variables s , CUR and V_CUR are process variables with implicit scenario and time indexes, their values are therefore written in plural form.

The following graph is the histogram of hedge error distribution for delta hedging the European put option with forward simulation. The exact value of the European put is 13.15 with a standard deviation of 0.84 for the hedge error.



The following code example is a ThetaML model for simulating hedge portfolio values backward in time, using Black-Scholes delta as the hedge strategy for the underlying hedge instrument .

```

1: model DeltaHedge_Backwards
2: % This model simulates backward in time the delta hedge of a
3: % European put option
4: import S           "Stock prices"
5: import CUR          "Discount factors"
6: import sigma        "Volatility of the stock price"
7: import r            "Risk-free interest rate"
8: import K            "Strike price for the European put option"
9: export Pi_CUR       "Hedging portfolio value"
10: export error        "Hedging error"
11:
12: T = 1 %time to maturity of the European put option
13: n = 252*T %loop length
14: error = Pi_CUR! - E(V_CUR!) %hedging error
15:
16: loop n
17:     % obtain the Black Scholes delta
18:     call BlackScholesDelta
19:     export K, sigma, r
20:     export S to S
21:     export T - @time to T
22:     import delta
23:     % update the hedging portfolio
24:     Pi_CUR = Pi_CUR! - delta * (S!*CUR! - S*CUR)
25:     theta T/n
26: end
27: % the hedging portfolio has the same value as V_CUR
28: Pi_CUR = V_CUR!
29: % at maturity T, option payoff is discounted to time 0
30: V_CUR = max(K - S,0) * CUR
31:
32: end

```

In the model `DeltaHedge_Backwards`, the European put option maturity T is set to 1 year (line 12), each year is assumed to have 252 trading days (line 13).



The model `DeltaHedge_Backwards` imports the stock prices `s` and the discount factors `CUR` simulated externally in the model `S_CUR_Processes` in the Chapter Example of Chapter 3. The two processes `s` and `CUR` automatically synchronize in model time with processes in current model. As such, `s` and `CUR` take their respective values at the corresponding model time. For example, `s` and `CUR` at line 24 take their respective values at the model time executed at that line of code. At line 30 they take their respective values at option maturity time T .

The model `DeltaHedge_Backwards` computes the hedging error based on the analytic solution of the Black-Scholes `delta`. The error is computed in a backwards fashion, such that `Pi_CUR` always contains the exact amount of money required for a perfect replication. This is conditioned by setting `Pi_CUR = V_CUR!` at the option maturity time (line 28) together with the no-arbitrage theorem that two portfolios having the same price in the future will have the same price today.

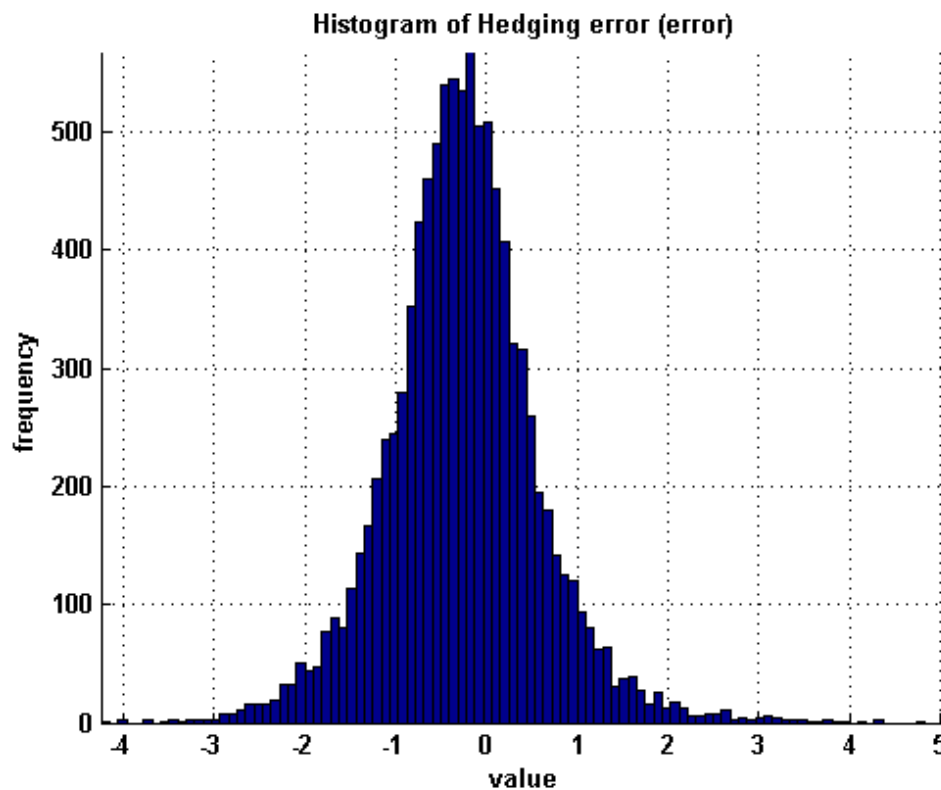
The backward-in-time style of hedging is facilitated with the ThetaML future operator `!.`. As such, we are able to fix the future hedging portfolio value to the option's discounted future payoffs (`Pi_CUR = V_CUR!` at line 28), respectively for all the simulation paths, and iterate backwards in time to arrive at current hedging portfolio values.

The hedging portfolio is assumed to be rebalanced at each trading day (line 24). The position taken in the hedging stock is `delta`, it is computed by calling the function `BlackScholesDelta` (line 18 - line 22) implemented in the model `BlackScholesDelta`.

The value of the hedging portfolio `Pi_CUR` is not deterministic and thus has a probability distribution at time 0. This distribution minus the option value $E(V_CUR!)$ is the hedging error (line 14). The distribution for the hedging error is shown in the graph below.



We discount to time 0 the processes v_{CUR} and s so that future cash flows are always seen in present value terms. The variables s , $_{CUR}$ and v_{CUR} are process variables with implicit scenario and time indexes, we therefore write their values in plural form.



As in the case of forward hedging, the value of the European put $E(v_{CUR})$, using Black-Scholes delta as the hedging strategy simulated backwards in time, is 13.15 with a standard deviation of 0.84 for the hedging error.



7.2.3 Variance Minimization by Hedging

Even though theory dictates that financial markets do not allow arbitrage and that markets are complete, even though this no-arbitrage-and-complete-market theory has led to various breakthroughs in the previous decades, there is in Thet-aML a unique technology that lies at the frontline of this derivatives research, by simply assuming that a real-world model for the underlying hedge instrument exists. Even though this view appears unconventional, we can obtain optimal hedging strategies for incomplete markets and in the presence of transaction costs.

Our optimal hedging strategies are computed by the function `Beta`. The `Beta(S, V)` function computes a variance minimal hedge based on the statistical properties of the underlying `s` and the instrument to be hedged `v`.

Our unique `Beta` function allows using Monte Carlo pricing for many more realistic market scenarios other than the basic ones assumed in the Black-Scholes economy. Moreover, even in a Black-Scholes economy, our `Beta` function gives the same results as the Black-Scholes delta. This is illustrated by the following Thet-aML code examples where the hedge instruments can be single- or multi-dimensional stochastic processes, or single or multiple underlyings.



7.2.3.1 Single Dimensional Stochastics

The case of beta hedging with single underlying, the underlying follows a single-dimensional stochastic process:

```

1: model BetaHedge_SingleUnderlying
2: % This model simulates the optimal hedge of an American put
3: % option, written on a single underlying stock;
4: % note that this hedge is variance optimal for any
5: % underlying process S
6: import S           "Stock prices"
7: import CUR          "Discount factors"
8: import sigma        "Volatility of the stock prices"
9: import r            "Risk-free interest rate"
10: import K            "Strike price for the American put option"
11: export Pi_CUR       "Hedging portfolio value"
12:
13: T = 1 % time to maturity of the put option
14: n = 252*T % loop length
15:
16: loop n
17:     % update the discounted hedging portfolio value
18:     Pi_CUR = Pi_CUR!
19:     - Beta(S!*CUR!, Pi_CUR!) * (S!*CUR! - S*CUR)
20:     % evaluate early exercise decisions
21:     if E(Pi_CUR!) < (K - S)*CUR
22:         Pi_CUR = (K - S)*CUR
23:     end
24:     % time passing of T/n time interval
25:     theta T/n
26: end
27: % hedging portfolio Pi_CUR has the same value as V_CUR
28: Pi_CUR = V_CUR!
29: % at maturity T, option payoffs discounted to time 0
30: V_CUR = max(K - S,0) * CUR
31:
32: end

```



In the model `BetaHedge_SingleUnderlying`, the time to maturity of the American put option T is given as 1 year at line 13, and we assume there are 252 trading days in one year (line 14).

The model `BetaHedge_SingleUnderlying` imports the stock prices s and the discount factors CUR simulated externally in the model `S_CUR_Processes` in the Chapter Example of Chapter 3. The two processes s and CUR automatically synchronize in model time with processes in current model. As such, s and CUR take their respective values at the corresponding model time. For example, s and CUR at line 19 take their respective values at the model time executed at that line of code. At line 30 they take their respective values at option maturity time T .

The hedging portfolio Pi_CUR is set up as having the same value as the discounted American put payoffs V_CUR . This is conditioned by setting $Pi_CUR = V_CUR!$ at the option maturity time T (line 28) together with the no-arbitrage theorem that two portfolios having the same price in the future should be priced the same today.

The portfolio Pi_CUR is updated at each T/n passing time interval (`theta T/n` at line 25), by investing an amount of $Beta(S!*CUR!, Pi_CUR!)$ in the stock s (line 19). The function $Beta(S!*CUR!, Pi_CUR!)$ computes the beta factor for its two arguments, where $S!*CUR!$ is the explanatory variable and $Pi_CUR!$ is the dependent variable. Assuming we are now at time t , the $Beta(S!*CUR!, Pi_CUR!)$ function estimates the relationship between $S!*CUR!$ and $Pi_CUR!$ conditional on the information at time t , where $s!$, $CUR!$ and $Pi_CUR!$ take respectively the values of s , CUR and Pi_CUR at the immediate next model time.

Since the option is American, the hedged portfolio continuation value is compared to the option's exercise value at each possible exercise date. Depending on the evaluation result, the hedged portfolio value is accordingly updated (line 21 - line 23).



We discount to time 0 the processes v_{CUR} and s so that future cash flows are always seen in present value terms. The variables s , CUR , Pi_{CUR} and v_{CUR} are process variables with implicit scenario and time indexes.



The case of beta hedging with multiple underlyings, the underlyings follow single-dimensional stochastic process:

```

1: model BetaHedge_MultipleUnderlyings
2: % This model simulates the optimal hedge of a European put
3: % basket option. The option is written on multiple
4: % underlyings with multiple strike prices; note: this hedge
5: % is variance optimal for any underlying process S
6: import S           "Stock prices"
7: import CUR          "Discount factors"
8: import K            "Strike prices for the American basket put"
9: export delta        "Hedge position"
10: export Pi_obs       "Hedging portfolio value"
11: export P            "European put basket option price"
12:
13: %P has the same expected value as Pi_CUR
14:
15: P = E(Pi_CUR!)
16: T = 1 % time to maturity of the basket option
17: n = 252*T % loop length, assuming 252 trading days
18:
19: loop n
20:     % we can store the values of Pi_CUR at each time step
21:     % in the variable Pi_obs for examinations
22:     Pi_obs = Pi_CUR!
23:     % array loop, loop through the elements of the arrays
24:     % S and delta!; delta! is defined after the array loop
25:     loop s, d : S, delta!
26:         % update portfolio values for the component stocks
27:         Pi_CUR = Pi_CUR! - d * (s!*CUR! - s*CUR)
28:     end
29:     % computes an array of beta factors
30:     delta = Beta(S!*CUR!,Pi_CUR!)
31:     % time passing of T/n time interval
32:     theta T/n
33: end
34: % Pi_CUR is set equal to V_CUR! at maturity T
35: Pi_CUR = V_CUR!

```



```

36: % loop through the arrays of stocks S and strikes K, and
37: % assign the discounted payoffs to V_CUR
38: loop s,k : S,K
39:     V_CUR = max(V_CUR!, (k - s) * CUR)
40: end
41:
42: V_CUR = 0 % cut-off value of option payoffs
43: delta = 0 % liquidate the stock positions at maturity
44:
45: end

```

In the model `BetaHedge_MultipleUnderlyings`, the European put option maturity T is set to 1 year (line 16), each year is assumed to have 252 trading days (line 17).

The model `BetaHedge_MultipleUnderlyings` imports the stock prices s and the discount factors CUR simulated externally. The two processes s and CUR automatically synchronize in model time with other processes in current model. As such, s and CUR take their respective values at the corresponding model time. For example, the processes s and CUR at line 30 take their respective values at the model time executed at that line of code. At line 38 and line 39 they take their respective values at the option maturity time T . Note in this model, the stock price processes s and the strike prices K are arrays.

At time 0, the European put option price P is set up as having the same expected value of the discounted hedging portfolio Pi_CUR . The values of the discounted hedging portfolio Pi_CUR are pinned at time T to the discounted payoffs of an exotic European put option v_CUR at line 35. By setting $Pi_CUR = v_CUR!$ at the option maturity time T , the no-arbitrage theorem dictates that the two portfolios Pi_CUR and v_CUR should have the same price today.

The payoffs of the exotic European put option v_CUR are defined by the code block from line 38 to line 40.

We elaborate more on this payoff feature:



The `loop ... end` block loops through the array of strikes and stock prices, and assigns to `v_CUR` a set of vanilla put option payoffs. If we expand the loop, the set of payoffs look as follows:

```
V_CUR = max(V_CUR!, (K[1] - S[1]) * CUR)
V_CUR = max(V_CUR!, (K[2] - S[2]) * CUR)
...
V_CUR = max(V_CUR!, (K[length(K)] - S[length(S)]) * CUR)
V_CUR = 0
```

where, for easier exposition, we use the square bracket `[]` to denote array indexing, and `length()` is the length of an array, `K[length(K)]` for example is the last array element of `K`. Since the payoff function uses the future referenced `v_CUR` (i.e. `v_CUR!`), the order of the code evaluation is backward in time. As such, we obtain first `v_CUR = 0` at line 42. Then input this value 0 into the payoff profile `v_CUR = max(0, (K[length(K)] - S[length(S)]) * CUR)`, note that `length(K) = length(S)` per the definition of loop in ThetaML. Since this is the payoff value future referenced by the `v_CUR!` before it, the payoff profile at index `length(K)-1` is thus `v_CUR = max(max(0, (K[length(K)] - S[length(S)]) * CUR), (K[length(K)-1] - S[length(S)-1]) * CUR)`. Continuing like this, we have the combined payoff for the exotic European put option:

```
V_CUR = max(max( ... max(0, (K[length(K)] - S[length(S)]) * CUR) ... , (K[2] - S[2]) * CUR), (K[1] - S[1]) * CUR)
```

Having fixed the maturity payoffs, the portfolio `pi_CUR` is then updated at each passing time interval T/n (line 27), by investing an amount of d in the component stock `s`. The amount of d is an element of the array `delta!` who access the beta factors computed for the set of discounted stocks `s! * CUR!` in the portfolio `pi_CUR!` (line 30). The array `delta!` is computed at each time step, starting from time 0 to time $T - (T/n)$.



Expanding the array loop line 25 - line 28, the code block look as follows:

```
Pi_CUR = Pi_CUR! - Beta[1]*(S[1]!*CUR! - S[1]*CUR)
Pi_CUR = Pi_CUR! - Beta[2]*(S[2]!*CUR! - S[2]*CUR)
...
Pi_CUR = Pi_CUR! - Beta[length(S)] * (S[length(S)]!*CUR! -
S[length(S)]*CUR)
Pi_CUR = Pi_CUR_next
```

Define $m = \text{length}(S)$, denote the portfolio value at the k -th loop as $\text{Pi_CUR}[k]$, the above set is put together as follows

```
Pi_CUR = ( ... ((Pi_CUR_next - Beta[m]*(S[m]!*CUR! - S[m]*CUR))
- Beta[m-1]*(S[m-1]!*CUR! - S[m-1]*CUR)) ... )
- Beta[1]*(S[1]!*CUR! - S[1]*CUR)
```

where, for easier exposition, we use the square bracket $[]$ to denote array indexing. The term Pi_CUR_next equals $v_CUR!$ at line 35. The above representation for Pi_CUR simply says that the portfolio value Pi_CUR is rebalanced for each stock by an amount of $\text{Beta}[k]$ for the component stock $S[k]$. The portfolio rebalancing process starts from time 0 to one step before the option maturity, $T - (T/n)$. At option maturity T , we liquidate the stocks by setting the array $\text{delta} = 0$ at line 43. A special note about the future operator “!” in the array loop: the operator “!” after delta (line 25) and Pi_CUR (line 27) access their respective *next* value at the same time step, while the operator “!” after s at line 30 accesses the stocks’ *future* values at the next time step.

We discount to time 0 the processes v_CUR , Pi_CUR and s so that future cash flows are always seen in present value terms. The variables s , CUR , Pi_CUR and v_CUR are process variables with implicit scenario and time indexes.



7.2.3.2 Multi-Dimensional Stochastics

The exact same ThetaML Script is applicable to stock prices following higher dimensional stochastic processes. The only difference is that the imported stock price process is multi-dimensional process simulated externally in a simulation model.



7.2.4 Static and Dynamic Hedging

To further illustrate the wide applicability of our `Beta` function, we provide, among others, the following code examples for static and dynamic portfolio hedging. The case of static portfolio hedging:

```

1: model Static_hedging
2: % This model applies our Beta function to static portfolio
3: % hedging of a Barrier option
4: import S           "Stock prices"
5: import CUR          "Discount factors"
6: export delta        "Hedge position"
7: export P_obs        "Hedged option price"
8:
9: % declare delta as a float array of length 20
10: % declare Vt_CUR as a float array of length 20
11: type delta float[20]
12: type Vt_CUR float[20]
13: % store the present values of P_CUR! in the variable
14: % P_obs for examinations of the results
15: P_obs = P_CUR!
16: % compute the beta factors for portfolio hedging
17: delta = Beta(Vt_CUR!, V_CUR!)
18: % loop through the arrays of delta and Vt_CUR
19: loop d, v : delta, Vt_CUR
20:     % set up the discounted hedging portfolio P_CUR at time
21:     % 0, using the hedging instruments in the array Vt_CUR
22:     P_CUR = P_CUR! - d * ( v! - E(v!) )
23: end
24: % the discounted hedging portfolio P_CUR initially
25: % has the same distribution as the discounted Barrier
26: % option values V_CUR at time 0
27: P_CUR = V_CUR!
28:
29: % the Barrier option to be hedged
30: fork
31:     Loop 52

```



```

32:         % one week passes
33:         theta 1/52
34:         % check whether or not the stock prices S violate
35:         % the barrier 120, for all Monte-Carlo paths
36:         if S > 120
37:             V_CUR = 0
38:         end
39:     end
40:     % at 1 year maturity, the discounted option payoff
41:     % is discounted to time 0
42:     V_CUR = max(100 - S, 0) * CUR
43: end
44:
45: % build up the hedge instruments
46: index = 1
47: loop 5
48:     % time passing of 1/5 year
49:     theta 1/5
50:     % the set of hedging instruments Vt_CUR maturing at
51:     % this model time
52:     Vt_CUR[index] = max(100 - S, 0) * CUR
53:     Vt_CUR[index+1] = max(120 - S, 0) * CUR
54:     Vt_CUR[index+2] = max(S - 100, 0) * CUR
55:     Vt_CUR[index+3] = max(S - 120, 0) * CUR
56:     % update the array index
57:     index = index + 4
58: end
59:
60: end

```

In the model `Static_hedging`, we define two arrays using the `type` keyword - `delta` and `Vt_CUR` - to hold respectively the beta factors and the hedging instruments.

The model `Static_hedging` imports the stock prices `s` and the discount factors `CUR` simulated externally in the model `S_CUR_Processes` in the Chapter Example of Chapter 3. The two processes `s` and `CUR` automatically synchronize in model time with processes in current model. As such, `s` and `CUR` take their respective



values at the corresponding model time. For example, `s` and `CUR` at line 42 take their respectively values at option maturity time 1.

At line 17, the hedge ratios `Beta(Vt_CUR!, V_CUR!)` are pre-computed conditional on current time information, the variables `Vt_CUR!` and `V_CUR!` are respectively values of the discounted hedge instruments and values of the discounted barrier option at time 0. The discounted hedging portfolio `P_CUR` adjust its values using the computed hedge ratios for the respective hedging instruments (line 22). The adjusting process is realized with the following loop block selectively copied from the model `Static_hedging`:

```
19: loop d, v : delta, Vt_CUR
22:     P_CUR = P_CUR! - d * ( v! - E(v!) )
23: end
27: P_CUR = V_CUR!
```

The above array loop can be expanded as follows:

```
P_CUR = P_CUR! - delta[1] * ( Vt_CUR[1]! - E(Vt_CUR[1]!) )
P_CUR = P_CUR! - delta[2] * ( Vt_CUR[2]! - E(Vt_CUR[2]!) )
...
P_CUR = P_CUR! - delta[length(delta)] * ( Vt_CUR[length(Vt_CUR)]! -
                                           E(Vt_CUR[length(Vt_CUR)]!) )
P_CUR = V_CUR!
```

where `length(delta)` must equal `length(Vt_CUR)` per the definition of array loop in `ThetaML`. We have used the square bracket `[]` to denote array indexing, for example, `delta[1]` means the first array element of the array `delta`, `Vt_CUR[length(Vt_CUR)]` is the last array element of the array `Vt_CUR`.

Since the discounted hedging portfolio `P_CUR` is referenced with the future operator “!”, the evaluation process of `P_CUR` is backward in time:

The discounted portfolio P_CUR initially has the same value distribution as the discounted barrier option ($P_CUR = V_CUR!$ at line 27). Define $n = \text{length}(\text{delta})$, k as the k -th element of the array of hedging instruments, the discounted portfolio P_CUR at the index element n is:

$$P_CUR = V_CUR! - \text{delta}[n] * (Vt_CUR[n]! - E(Vt_CUR[n]!))$$

The discounted portfolio P_CUR at index $n-1$ is:

$$P_CUR = (V_CUR! - \text{delta}[n] * (Vt_CUR[n]! - E(Vt_CUR[n]!))) - \text{delta}[n-1] * (Vt_CUR[n-1]! - E(Vt_CUR[n-1]!))$$

Repeating this procedure, the discounted portfolio after looping through all the elements of the array is represented as:

$$P_CUR = (\dots ((V_CUR! - \text{delta}[n] * (Vt_CUR[n]! - E(Vt_CUR[n]!))) - \text{delta}[n-1] * (Vt_CUR[n-1]! - E(Vt_CUR[n-1]!))) \dots - \text{delta}[1] * (Vt_CUR[1]! - E(Vt_CUR[1]!)))$$

Mathematically, this is equivalent to

$$P_CUR = V_CUR! - \sum_{k=1}^n \text{delta}_k (Vt_CUR[k]! - E(Vt_CUR[k]!)),$$

where P_CUR and V_CUR are respectively the discounted hedging portfolio and the discounted barrier option, the term $Vt_CUR[k]$ is the k -th element of the array of discounted hedging instruments. The term delta_k denotes the hedge ratio for instrument $Vt_CUR[k]$. After the discounted portfolio is initially hedged with a delta_k amount in the k -th instrument, we hold this position throughout the lifetime of the hedged barrier option.

The $\text{Beta}(Vt_CUR!, V_CUR!)$ function at line 17 takes two arguments, the hedging instruments $Vt_CUR!$ as the explanatory variables and the barrier option value $v_CUR!$



$CUR!$ as the dependent variable. Both the two function arguments involve the future operator “!”, their future values are determined in the following.

The `fork ... end` block from line 30 to line 43 runs the process of the discounted barrier option v_CUR , the `loop ... end` block from line 47 to line 58 builds the discounted hedge instruments vt_CUR . The `fork ... end` statement and the `theta` command enable the two processes v_CUR and vt_CUR run virtually in parallel in model time. This is as if the second `loop ... end` block (line 47 - line 58) were implicitly forked.

In the first `loop ... end` block (line 31 to line 39), we check every week ($1/52$) whether the stock price is above the barrier or not, if it is, the barrier option is knocked out; otherwise, the option remains alive. At year 1 ($52 * (1/52)$), we reach the maturity of the barrier option and assign the discounted payoffs to v_CUR .

In the loop for the hedging instruments (line 47 to line 58), at time $1/5$, the first 4 hedging instruments mature and are given their respective discounted payoff values. After another $1/5$ time - at time $2/5$, another 4 of the hedging instruments mature and are assigned their respective discounted payoffs. Continuing this procedure, at time 1 ($5 * (1/5)$), the last 4 hedging instruments get their respective discounted payoffs.

We discount to time 0 the processes v_CUR and vt_CUR so that future cash flows are always seen in present value terms. The discount factors CUR decay at a constant risk-free rate r , as such all the variable processes at a certain time point are discounted by the same discount factor maturing at that time point. The variables S , CUR , vt_CUR and v_CUR are process variables with implicit scenario and time indexes.

Computationally, we evaluate first the payoff functions at maturity time 1, then updating each of the processes according to their respective process features, continuing this way backwards to arrive their respective values at current time.



The case of dynamic portfolio hedging:

```

1: model Dynamic_hedging
2: % This model applies our Beta function to dynamic portfolio
3: % hedging
4: import S           "Stock prices"
5: import CUR          "Discount factors"
6: export Pi_obs       "Dynamically hedged portfolio values"
7: export V_obs        "Unhedged barrier option prices"
8:
9: % store the present values of the barrier option in
10: % V_obs, and store the values of the hedging portfolio
11: % in the variable Pi_obs for examinations of results
12: V_obs = V_CUR!
13: Pi_obs = Pi_CUR!
14:
15: % dynamic portfolio hedging
16: fork
17:   loop 252
18:     % rebalance the discounted portfolio Pi_CUR
19:     Pi_CUR = Pi_CUR! - Beta(S!*CUR!, Pi_CUR!)
20:               * (S!*CUR! - S*CUR)
21:     % 1 trading day passes
22:     theta 1/252
23:   end
24: end
25:
26: % barrier option
27: fork
28:   loop 52
29:     % one week passes
30:     theta 1/52
31:     % check if the stock prices S violates
32:     % the barrier 120
33:     if S > 120
34:       % if the barrier is violated, the discounted barrier
35:       % option V_CUR is knocked out and the discounted
36:       % hedging portfolio Pi_CUR has zero values, for the

```



```

37:         % violated simulation paths
38:         V_CUR = 0
39:         Pi_CUR = 0
40:     end
41: end
42: % the discounted hedging portfolio Pi_CUR has the same
43: % distribution as the discounted barrier option V_CUR
44: % at the option maturity time
45: Pi_CUR = V_CUR!
46: V_CUR = max(100 - S,0) * CUR
47: end
48:
49: end

```

The model `Dynamic_hedging` is a good example for the use of the `fork ... end` statement. The body of the model is formed by two `fork ... end` blocks that run virtually in parallel in model time. The first `fork ... end` block (line 16 - line 24) runs the process of a discounted hedging portfolio `Pi_CUR`, the second `fork ... end` block (line 27 - line 47) runs the process of a barrier option `V_CUR` and checks the barriers for the discounted barrier option `V_CUR` and the discounted hedging portfolio `Pi_CUR`.

The model `Dynamic_hedging` imports the stock prices `s` and the discount factors `CUR` simulated externally in the model `S_CUR_Processes` in the Chapter Example of Chapter 3. The two processes `s` and `CUR` automatically synchronize in model time with processes in current model. As such, `s` and `CUR` take their respective values at the corresponding model time. For example, `s` and `CUR` at line 46 take their respective values at option maturity time 1.

In the first `fork ... end` block (line 16 - line 24), the discounted portfolio `Pi_CUR` is rebalance at a constant time interval $1/252$. The portfolio rebalancing process is realized with the `loop ... end` loop (line 17 - line 23). In the loop, at each passing time of $1/252$, the discounted portfolio values are updated by the amount of



$\text{Beta}(S!*\text{CUR!}, \text{Pi_CUR!})$ in the underlying stocks s . The Beta function computes the factor loadings for the discounted stock prices $S!*\text{CUR!}$ with respect to the discounted portfolio values Pi_CUR! , conditional on the information available at that passing model time.

In the second `fork ... end` block (line 27 - line 47), we check at every $1/52$ time interval whether the stock s has breached the barrier 120 or not, if it has, the discounted barrier option v_CUR is knocked out and the discounted hedging portfolio value Pi_CUR is set to 0; this is done for all the Monte-Carlo simulation paths. At time 1 ($52*(1/52)$), the barrier option has the discounted payoffs $\max(100 - S, 0) * \text{CUR}$ (line 46), and the hedging portfolio Pi_CUR gets the same discounted payoffs ($\text{Pi_CUR} = v_CUR!$ at line 45, where $v_CUR!$ takes the value of v_CUR at line 46).

The variables s , CUR , Pi_CUR and v_CUR are process variables with implicit scenario and time indexes.

The processes v_CUR and s are discounted to time 0 so that we always talk about future cash flows in present value terms. The discount factors CUR decay at a constant risk-free rate r , as such all the variable processes at a certain time point are discounted by the same discount factor maturing at that time point.

Computationally, both of the payoff functions at maturity time 1 are evaluated first, then each of the processes are updated according to their respective process features. Continuing backwards in time, we arrive at their respective values at time 0.



7.2.5 Transaction Costs

Our unique `Beta` function also computes optimal hedging strategies in the presence of transaction costs. We show two such code examples below - one for portfolio rebalancing at constant time interval and one for portfolio rebalancing only when the underlying position changes are above a certain barrier.

The case of portfolio rebalancing at constant time interval:

```

1: model BetaHedge_TCost_ConstInterval
2: % This model applies our Beta function to hedge a European
3: % put option in the presence of transaction cost, the
4: % portfolio is rebalanced at constant time interval
5: import S           "Stock prices"
6: import CUR         "Discount factors"
7: import K           "Strike price for the European put option"
8: import T           "Time to maturity"
9: export Pi_CUR       "Option value"
10: export Pif_CUR     "Hedged option value"
11: export Error       "Hedging error"
12: export delta       "Hedge position"
13:
14: n = 252             % loop length
15: kappa = 0.01        % level of transaction cost
16: % the forward hedged portfolio Pif_CUR initially has the
17: % same expected value as the discounted backward hedging
18: % portfolio Pi_CUR
19: Pif_CUR = E(Pi_CUR!)
20: delta_old = 0       % old delta, backward hedging
21: delta_old_f = 0     % old delta, forward hedging
22: % initialize the discounted old stock values Sold_CUR to
23: % the discounted current stock values S*CUR
24: Sold_CUR = S*CUR
25:
26: loop n
27:     % optimally update the discounted backward-hedging
28:     % portfolio Pi_CUR considering transaction costs

```



```

29:     Pi_CUR = Pi_CUR! - delta!*(S!*CUR! - S*CUR)
30:         + kappa * abs( (delta_old! - delta!)*S*CUR)
31:     % limit the position in stock S to [-1, 0] since the
32:     % hedged option is a put option
33:     delta = min(0, max(-1, Beta(S!*CUR!,Pi_CUR!)))
34:     % update delta_old which is computed in the
35:     % next model time
36:     delta_old = delta!
37:     % time passing of T/n time interval
38:     theta T/n
39:     % if the position taken in the stock S has changed from
40:     % the previous one, rebalance the portfolio
41:     if (abs(delta - delta_old_f) >= 0.0) & @time < T
42:         Pif_CUR = Pif_CUR + delta * (S*CUR - Sold_CUR)
43:             - kappa * abs( (delta_old_f - delta)*S*CUR)
44:         % update the old discounted stock price
45:         Sold_CUR = S*CUR
46:         % update the old delta
47:         delta_old_f = delta
48:     end
49: end
50: % at option maturity, for forward hedged portfolio: unwind
51: % the position in stocks, including transaction costs
52: Pif_CUR = Pif_CUR - kappa * abs( (delta_old_f - 0)*S*CUR)
53: % liquidate stock positions at option maturity
54: delta = 0
55: % at option maturity, for backward hedging portfolio: set
56: % the hedging portfolio Pi_CUR to the values of discounted
57: % option payoffs V_CUR
58: Pi_CUR = V_CUR!
59: % discounted option maturity payoffs
60: V_CUR = max(K - S,0) * CUR
61: % hedging error
62: Error = V_CUR - Pif_CUR
63:
64: end

```

In the model `BetaHedge_TCost_ConstInterval`, the European put option maturity T is imported as an input parameter, each year is assumed to have 252 trading days (line 14). Transaction cost is a constant κ at line 15.



Hedging in the presence of transaction cost is based on the idea that, at option maturity time, we can fix the hedging portfolio value to the known option maturity payoff. By the no-arbitrage theorem, two portfolios having the same cash flows in the future should have the same price today. By pinning the hedging portfolio value to the option maturity payoff, we are able to hedge the portfolios backwards in time with an optimal hedge ratio for each portfolio rebalancing date. The optimal hedge ratio is computed based on the value of the hedging portfolio in the presence of transaction cost, conditional on the filtration of current stock prices. Continuing iteratively back to the current time, the so-hedged two portfolios should have about the same price, depending on the optimality of the hedge ratios.

In ThetaML, the `Beta` function is thus defined such that it is variance optimal for almost any underlying process.

The model `BetaHedge_TCost_ConstInterval` imports the stock prices `s` and the discount factors `cur` simulated externally in the model `S_CUR_Processes` in the Chapter Example of Chapter 3. The two processes `s` and `cur` automatically synchronize in model time with processes in current model. As such, `s` and `cur` take their respective values at the corresponding model time. For example, `s` and `cur` at line 29 takes their respective values at the model time executed at that line of code. At line 60, `s` and `cur` take their respective values at option maturity time T . The backward hedging portfolio `Pi_CUR` is set up in order to obtain better estimates for hedge ratios that are computed using the `Beta` function.

At maturity time T , the backward-hedging portfolio `Pi_CUR` has the same value distribution as the discounted European put option payoffs (line 58 - line 60). Going backwards in time, the portfolio `Pi_CUR` is dynamically hedged using an optimal hedge ratio obtained from the `Beta(S!*CUR!,Pi_CUR!)` function; in the meantime, the hedging portfolio values are adjusted by an amount of transaction costs at `kappa` of the changes in stock investments (line 29 - line 30). The `Beta(S!*CUR!,Pi_CUR!)` function takes two arguments, the first argument



$s! \cdot \text{CUR}!$ is the explanatory variables and the second argument $\text{Pi_CUR}!$ is the dependent variable. Both of the two function arguments assess their future values via the future operator “!”. The $\text{Beta}(s! \cdot \text{CUR}!, \text{Pi_CUR}!)$ function computes an optimal factor loading based on the hedging portfolio value Pi_CUR in the presence of transaction costs, conditional on the then passing model time.

Line 33 limits the computed Beta values to an interval of -1 to 0, since the hedged option is a put option. The $\text{delta}!$ at line 29 and line 30 accesses the next delta value computed in line 33. The $\text{delta_old}!$ at line 30 accesses the next delta_old in line 36. As such, the term $(\text{delta_old}! - \text{delta}!)$ computes changes between the stock positions taken at the next and current model time.

The hedged option value Pif_CUR is set up initially to have the same expected value as the backward-hedging portfolio value Pi_CUR (line 19) at time 0. With time passing, the forward hedged option value Pif_CUR is dynamically updated with position delta in the underlying s (line 42), only when the position has changed from the previous portfolio update (line 41). The delta in line 41, line 42 and line 43 is computed in line 33 based on the backward-hedging portfolio Pi_CUR . As such, the term $\text{abs}(\text{delta_old_f} - \text{delta})$ computes change(s) of stock positions taken between previous and current model time. At option maturity, when we liquidate the stock positions, we adjust the forward hedging portfolio Pif_CUR by the corresponding amount of the transaction cost (line 52).

The variables s , CUR , Pi_CUR and Pif_CUR are process variables with implicit scenario and time indexes.

The processes v_CUR and s are discounted to time 0 so that we always talk about future cash flows in present value terms. The discount factors CUR decay at a constant risk-free rate r , as such all the variable processes at a certain time point are discounted by the same discount factor maturing at that time point.



The case of portfolio rebalancing only when changes in the underlying positions are above a certain barrier:

```

1: model BetaHedge_TCost_PositionBarrier
2: % This model applies our Beta function to hedge a European
3: % put option in the presence of transaction cost; the
4: % portfolio is rebalanced only when the positions in
5: % underlying hedge instruments change more than some level
6: import S           "Stock prices"
7: import CUR         "Discount factors"
8: import K           "Strike price for the European put"
9: import T           "Time to maturity"
10: export Pi_CUR      "Option value"
11: export Pif_CUR     "Hedged option value"
12: export Error       "Hedging error"
13: export delta       "Hedge ratios"
14:
15: n = 252           % loop length
16: kappa = 0.01      % level of transaction cost
17: % the forward hedged portfolio Pif_CUR initially has the
18: % same expected value as the discounted backward hedging
19: % portfolio Pi_CUR
20: Pif_CUR = E(Pi_CUR!)
21: delta_old = 0      % old delta, backward hedging
22: delta_old_f = 0    % old delta, forward hedging
23: % initialize the discounted old stock values Sold_CUR to
24: % the discounted current stock values S*CUR
25: Sold_CUR = S*CUR
26:
27: loop n
28:     % optimally update the discounted backward-hedging
29:     % portfolio Pi_CUR considering transaction costs
30:     Pi_CUR = Pi_CUR! - delta!*(S!*CUR! - S*CUR)
31:             + kappa * abs( (delta_old! - delta!)*S*CUR)
32:     % limit the position in stock S to (-1, 0), since the
33:     % hedged option is a put option
34:     delta = min(0, max(-1, Beta(S!*CUR!,Pi_CUR!)))
35:     % update delta_old which is computed in the

```



```

36:         % next model time
37:         delta_old = delta!
38:         % time passing of T/n time interval
39:         theta T/n
40:         % if the position changes in the stock S is above 0.05,
41:         % rebalance the portfolio Pif_CUR
42:         if (abs(delta - delta_old_f) >= 0.05) & @time < T
43:             Pif_CUR = Pif_CUR + delta * (S*CUR - Sold_CUR)
44:                 - kappa * abs( (delta_old_f - delta)*S*CUR)
45:             % update the old discounted stock price
46:             Sold_CUR = S*CUR
47:             % update the old delta
48:             delta_old_f = delta
49:         end
50:     end
51:     % at option maturity, for forward hedged portfolio: unwind
52:     % the position in stocks, including transaction costs
53:     Pif_CUR = Pif_CUR - kappa * abs( (delta_old_f - 0)*S*CUR)
54:     % liquidate the stock positions at option maturity
55:     delta = 0
56:     % at option maturity, for backward hedging portfolio: set
57:     % the hedging portfolio Pi_CUR to the values of discounted
58:     % option payoffs V_CUR
59:     Pi_CUR = V_CUR!
60:     % discounted option maturity payoffs
61:     V_CUR = max(K - S, 0) * CUR
62:     % hedging error
63:     Error = V_CUR - Pif_CUR
64:
65: end

```

In the model `BetaHedge_TCost_PositionBarrier`, the European put option maturity T is imported as an input parameter, each year is assumed to have 252 trading days (line 15). Transaction cost is a constant κ at line 16.

The model `BetaHedge_TCost_PositionBarrier` imports the stock prices s and the discount factors CUR simulated externally in the model `S_CUR_Processes` in the Chapter Example of Chapter 3. The two processes s and CUR automatically synchro-



nize in model time with processes in current model. As such, s and cur take their respective values at the corresponding model time. For example, s and cur at line 30 take their respective values at the model time executed at that line of code. At line 61, s and cur take their respective values at option maturity time T .

The ThetaML code statements are very similar to those in the model `BetaHedge_TCost_ConstInterval`. The only difference lies in line 42, where the hedged portfolio value P_{if_CUR} is adjusted only when the position changes are above the level of 0.05. This reduces the hedging frequency and thus transaction costs. This is more reasonable in cases where the volatilities of the underlying prices are small.

The backward-hedging portfolio value P_{i_CUR} is fixed at maturity time T to the European put option payoff (line 59), then iteratively going backwards in time, the portfolio P_{i_CUR} is dynamically hedged using an optimal hedge ratio obtained from the `Beta` function; in the meantime, the portfolio values are adjusted by an amount of transaction costs at κ of the changes in stock investments (line 30 - line 31).

Line 34 limits the computed `Beta` values to an interval of -1 to 0 for put option. The `delta!` at line 30 and line 31 accesses the next `delta` value computed in line 34. The `delta_old!` at line 31 accesses the next `delta_old` in line 37. As such, the term `abs(delta_old! - delta!)` computes changes in stock positions between next and the current model time.

The hedged option value P_{if_CUR} is set up initially to have the same expected value as the backward-hedging portfolio value P_{i_CUR} (line 20). With time passing, the portfolio value P_{if_CUR} is dynamically updated with position `delta` in the underlying s (line 43), only when the position has changed from the previous update by a factor of 0.05 (line 42). The `delta` in line 42, line 43, line 44 and line 48 are computed in line 34. As such, the term `abs(delta_old_f - delta)`



computes change(s) of stock positions between the previous and current one(s). At option maturity, when we liquidate the stock positions, we adjust the forward hedging portfolio Pif_CUR by the corresponding amount of the transaction cost (line 53).

The variables S , CUR , Pi_CUR and Pif_CUR are process variables with implicit scenario and time indexes.

The processes v_{CUR} and s are discounted to time 0 so that we always talk about future cash flows in present value terms. The discount factors CUR decay at a constant risk-free rate r , as such all the variable processes at a certain time point are discounted by the same discount factor maturing at that time point.

This completes our tutorial “Hedging in ThetaML”.



8 ThetaML Tips and Tricks

When working with ThetaML, there are various techniques which can be used to improve the speed and accuracy of Monte-Carlo simulations with ThetaML models. This chapter explains the most important ones.



8.1 Nested if Improves Speed

- Avoid evaluating conditional functions that returns zero

The most costly operation in pricing options is the evaluation of conditional stochastic functions such as E and $Beta$. The time complexity of such an operation is at least of order $O(n^2)$ with n as the number of Monte-Carlo paths. In many cases, we can selectively reduce the simulated paths to the most relevant ones before doing any computations with the conditional functions E and $Beta$.

As an example, consider the following `if ... end` block:

```
if E(V_CUR!) < (K - S)*CUR
    V_CUR = (K - S)*CUR
end
```

Since we know in advance that $E(V_CUR!)$ must be greater than zero, we restrict the whole computations to those simulation paths where $(K - S)*CUR$ is greater than zero.

```
if 0 < (K - S)*CUR
    if E(V_CUR!) < (K - S)*CUR
        V_CUR = (K - S)*CUR
    end
end
```



8.2 Reducing Variance by Hedging Improves Convergence

- The `Beta` function reduces the variance of simulated option prices thus increases the accuracy of price estimates.

The following two models `American` and `American_hedged` have similar runtimes but the model `American_hedged` returns option values of higher precision as a result of using the `Beta` function.

```

model American
% Model American simulates prices for an American put option
    import S      "Stock prices"
    import CUR     "Discount factors"
    import sigma  "Volatility"
    import r      "Interest rate"
    import K      "Strike price"
    export P      "Option value"

    % current option value
    P = E(V_CUR!)
    % time to maturity of the option
    T = 1
    % n loops, assuming daily exercise interval and
    % 252 trading days in a year
    n = 252*T
    loop n
        % evaluate optimal exercise conditions, compare estimated
        % discounted hold values E(V_CUR!) with discounted
        % intrinsic values, for all Monte-Carlo paths
        if E(V_CUR!) < (K - S)*CUR
            V_CUR = (K - S)*CUR
        end
        % time passing of T/n
        theta T/n
    end
    % at maturity, option payoffs are discounted to time 0
    V_CUR = max(K - S, 0)*CUR
end

```



```

model American_hedged
% Model American_hedged simulates the optimal hedge of an
% American put option; note that this hedge is variance
% optimal for any underlying process S
    import S      "Stock prices"
    import CUR    "Discount factors"
    import sigma  "Volatility"
    import r      "Interest rate"
    import K      "Strike price"
    export P      "Option value"

% current option value
P = E(Pi_CUR!)
% time to maturity
T = 1
%n loops
n = 252*T
loop n
    % hedge the discounted portfolio Pi_CUR using Beta function
    Pi_CUR = Pi_CUR! - Beta(S!*CUR!,Pi_CUR)*(S!*CUR! - S*CUR)
    % evaluate optimal exercise conditions, compare estimated
    % discounted hold values E(Pi_CUR!) with discounted
    % intrinsic values, for all Monte-Carlo paths
    if E(Pi_CUR!) < (K - S)*CUR
        Pi_CUR = (K - S)*CUR
    end
    % time passing of T/n
    theta T/n
end
% at option maturity, the hedged portfolio has the same value
% distribution as the discounted option payoffs
Pi_CUR = V_CUR!
% at maturity, the option payoff is discounted to time 0
V_CUR = max(K - S,0)*CUR
end

```



8.3 Adding Control Variables Can Improve Accuracy

- Using additional information available, we can significantly improve the numerical accuracies of conditional expected values such as E and $Beta$

In the model `European_S_control`, the underlying S is simulated with no drift, as such we can use its price at time 0 as an estimate of its expected value at time 1. We can then use this knowledge to compute a better estimate P for the expected value of V_CUR as follows:

```

model European_S_control
% Pricing a European put option with S as control variable
    import S      "Stock prices"
    import CUR     "Discount factors"
    import K       "Option strike price"
    export P       "Option price"

    % use S as control variable for better price estimates P
    P = V_CUR! - Beta(S_control!,V_CUR!) * (S_control! - S)

    % 1 time unit passes
    theta 1

    % S_control has expected value S
    S_control = S
    % at maturity, the option payoff is discounted to time 0
    % S is already discounted since it is simulated with no drift
    V_CUR = max(K*CUR - S,0)
end

```



We can even go a little further and include other knowledge like the closed-form solution of a European option price and use it to estimate an American option price:

```
% Pricing an American put option with a European put option as
% control variable
model American_Vcontrol
    import S      "Stock prices"
    import CUR    "Discount factors"
    import K      "Option strike price"
    import sigma  "Volatility"
    import r      "Risk-free rate"
    export P      "Option price"

    % use European option V_control as control variable
    % V_control has expected value V_ref
    P = V_CUR! - Beta(V_control,V_CUR) * (V_control! - V_ref!)
    % V_ref is computed using the closed form formula for the
    % European put option; the function arguments for myBlsprice
    % are respectively: stock price, strike price, volatility,
    % interest rate, option maturity, and option type 'call';
    % the option price V_ref is discounted to time 0 by CUR, this
    % is to be consistent with its simulated counterpart V_control
    V_ref = myBlsprice(S, K, sigma, r, 1, 0)*CUR
    %loop 52 times
    loop 52
        % time passing of one week
        theta 1/52
        % early exercise evaluation: compare the expected discounted
        % hold values E(V_CUR!) with discounted option intrinsic
        % values, for all the Monte-Carlo simulation paths
        if E(V_CUR!) < (K - S)*CUR
            V_CUR = (K - S)*CUR
        end
    end
    % at maturity 1, the control variable V_control has the same
    % discounted payoffs as the American put option V_CUR
    V_control = V_CUR!
    % at maturity 1, the payoff is discounted to time 0
    V_CUR = max(K - S,0)*CUR
end
```



8.5 Avoiding Direct Assignment of Expected Values Improves Accuracy

- We can improve the accuracy of option price estimates by avoiding direct assignments of the expected values from functions E and $Beta$

The expected value function E generates approximation errors and should only be used when a (statistical) expected value is required. For example, in a compound option, it is advisable not to use the E function for both decisional and computational steps:

```
model compound
...
theta 1
%time 1 outer option payoffs, E(V_inner) is the expected value
%of the inner option conditional on time 1 information
V = max( E(V_inner!)- K_outer, 0 )

theta 1
%time 2 inner option payoffs
V_inner = max(S - K_inner,0)

end
```



Instead, the decisional and computational parts should be separated as follows:

```
model compound
...

theta 1
% at time 1, evaluate the outer option payoffs
% E(V_inner!) is the expected value of the inner option,
% conditional on time 1 information
if E(V_inner!) - K_outer > 0
    V = V_inner! - K_outer
else
    V = 0
end

theta 1
% at time 2, inner option payoffs
V_inner = max(S - K_inner, 0)
end
```



8.6 Keep Export Variables Unique

- Export variables serve to report (return) results of a model. If they are also used to store temporary results, the exported results will get clogged. This situation can be avoided by keeping assignments to export variables unique.

In the following code, the first instance of `x` is a temporary result that gets incorrectly reported:

```
model test
    export x % results of x get clogged
    ...
    x = x/(n-1) % x has the temporary results
    x = x + S/n
    ...
end
```

Storing the intermediate results in a temporary variable keeps the reporting of `x` unique:

```
x_tmp = x % store x in a temporary variable x_tmp
x_tmp = x_tmp/(n-1) % doing computations with x_tmp
x_tmp = x_tmp + S/n % doing computations with x_tmp
x = x_tmp % assign x_tmp to x and report x
```





References

Dirnstorfer, S., 2004, On the Representation of Trading Strategies and Financial Portfolios, Intelligent Finance - Convergence of Financial Mathematics with Technical and Fundamental Analysis, Proc. 1st Int. Workshop on Intelligent Finance (IWIF-I), Melbourne, Australia, ISBN 187685118X, page 131 - 143.

Dirnstorfer, S., 2006, Multiscale Calculus with Applications in Quantitative Finance, PhD Thesis, TU München, Germany, Fakultät Für Informatik.

Dirnstorfer, S., A. J. Grau, 2008, Accelerated Option Pricing in Multiple Scenarios, Computational Engineering, Finance, and Science (cs.CE). Source: <http://arxiv.org/abs/0807.5120>

Dirnstorfer, S., A. J. Grau, Computer Aided Finance: Another Journey in the Quest for the Holy Grail of Financial Engineering, Technical Article, WILMOTT magazine, November 2008, page 68 - 73.

Dirnstorfer, S., A. J. Grau, and R. Zagst, 2006, Moving Window Asian Options: Sparse Grids and Least-Squares Monte Carlo, Working Paper.

Grau, A. J., 2008, Applications of Least-Squares Regressions to Pricing and Hedging of Financial Derivatives, Dissertation, Technische Universität München. Source: <http://nbnresolving.de/urn/resolver.pl?urn:nbn:de:bvb:91-diss-20071212-635889-1-9>

Thetaris: When Upstream Is Faster, Interview Article, WILMOTT magazine, September 2008, page 18 - 19.

Internet Resources: http://wiki.thetaris.com/index.php/Main_Page



Index

.
 .thetagram, 28
 .thetaml, 28
 .thetas, 28

@

@dt, 25, 70-71, 95, 110-111 112, 114, 117, 118
 @matlab, 83, 85, 94, 98-99, 127, 151
 @noBackward, 137-138
 @noExternalFunction, 137-138
 @noExternalModule, 137-138
 @noFork, 138
 @noInfLoop, 138
 @noRegression, 137
 @notheta, 137-138
@scenarioIndex, 26, 48
@scenarioSize, 26, 48
@time, 26, 48, 70, 114, 170, 173, 194

A

American option, 159, 164, 206
 array, 66-68, 73-74, 109, 129, 146, 185
 array loop, 73, 180, 183, 187
 Asian option, 94
 assert, 133-135, 139
 assignments, 53, 207, 209
 Assignments, 53, 145

B

Bermudean Option, 157



Beta function, 149, 153, 164, 166, 176, 185, 190, 192-193, 195, 203-204
Beta(), 25, 48, 108, 137
Black-Scholes delta, 167-174
Boolean, 47, 88, 126, 143

C

call, 47, 76-77, 79-81, 85, 94, 98-99, 148, 170, 173
chronological, 12, 14-17, 19, 24, 120
Comments, 52
complex stepping object, 82, 85, 87
compound option, 161, 207
computational order, 12, 14, 15, 158, 163
configuration file, 28, 39, 149-150
constraints, 132, 134-136, 139

D

data type, 49, 141, 143
Delta Hedging, 167
Dimension filter, 32
discount factor, 95, 99-100, 112, 114, 116-119, 154, 192
Domain Specific Language, 12, 43-44

E

E(), 16-17, 25, 48, 105-106, 123, 137
editor, 37-38, 40
Enum, 47, 128, 143
European option, 155, 206
export, 46-51, 55, 76-81, 85, 88, 94, 99, 110, 129, 136, 142, 146, 148, 150

F

files, 28, 29, 127, 149



`float`, 67-68, 129, 185
`fork ... end`, 17-18, 20-21, 24, 58-62, 72, 77, 99-100, 112-113
 future operator `!`, 14-16, 24, 51, 81, 101, 102-104, 106, 108, 120, 122-123, 156,
 183, 188-189, 196

G

Geometric Brownian motion, 38, 91, 115-116

H

hedging, 166-167, 170-178, 180-181, 185-200

I

`if ... else ... end`, 63
`implements`, 132-136
 implicit `fork ... end`, 77
`import`, 46-51, 55, 76-81, 85, 88, 94, 99, 126, 127-128, 133-134, 139, 142, 150
`indexOf()`, 73, 146
`interface`, 132-136, 138-139
 interfaces, 18, 131, 134

L

language constraints, 137
 Least Squares Monte Carlo, 17
`length()`, 68, 109, 149
`load`, 149-150
`loop ... end`, 25, 69-70, 72, 100-101, 112-113, 182, 189, 191
`loop inf ... end`, 24, 71-72, 112-114
 LSMC. *See* Least Squares Monte Carlo

M

Markov states, 92, 98



Matlab, 82, 151
Matlab function, 82-84
Matlab stepping object, 85-87
model, 46, 50-52
Model arguments, 46
model file, 28-29, 36, 39-40
model time, 16-21, 24, 26, 30-31, 54-56, 58-59, 61, 77, 82, 95, 101, 111, 113, 120, 156-157, 160
multiple arrays, 74

N

naming convention, 47

O

Open in Excel, 32-33

Open in Matlab, 32

P

portfolio hedging, 166, 185, 190

post conditions, 12, 18

Pre- and post-conditions, 12, 18

pre- and post-processing, 141

process variables, 101, 118, 156

project, 35-36

Project Explorer, 28, 35-36

R

[run](#), 149-150

Runtime parameters, 29, 40



S

Sample Project, 28
stepping function, 82, 85, 90, 94, 96, 98, 101
submodel, 76

T

`theta`, 16, 20-21, 24, 54-56, 58-62, 70-72, 95, 106, 110, 112-113
Theta Orchestrator, 128
Theta Suite, 27
Theta Suite Result Explorer, 30-32, 47
Thetagram, 10-11, 28
ThetaML, 9-19
ThetaML Configurator, 28, 32
Time-step filter, 31
Tips and Tricks, 201
transaction costs, 193
`type`, 49, 67-68, 126-129, 185
type extraction, 125, 143

V

Value assertions, 139
variables, 46-48, 120, 125
variance optimal hedge, 164

W

`workflow`, 142, 150



ThetaML is a modeling language developed by Thetaris. It describes financial products in a simple and general way. ThetaML introduces tools for intuitive model design, powerful analyses and automated productions. It focuses on financial product structural features and abstracts these structural features from model processes and numerical details.

As domain specific language, ThetaML bundles together functional and procedural programming styles. Important features of this language include:

- Domain specific language for financial contract design
- Programming in chronological order and computational order
- Implicit handling of scenario- and time- indices
- Virtual multi-threading of paralleled models
- Built-in conditional expectations of financial variables
- Pre- and post-conditions on models to ensure model correctness

This book documents the language syntax of ThetaML. It starts with a summary of the language features, followed by a chapter on ThetaML language syntax. The ThetaML type system, interfaces and workflows are detailed in later chapters. There are many code examples to help understand the language commands and functions. Two tutorials further apply ThetaML to pricing and hedging financial contracts. The final chapter offers many tips and tricks for more efficient use of ThetaML in financial settings.

