foveSIMM
a formally verified Standard Initial Margin Model

fovefi ltd[*]

January 13, 2020

Abstract

We introduce **foveSIMM** — a formally verified Standard Initial Margin Model (SIMM). It re-implements **openSIMM** in Isabelle, a powerful proof assistant. With **foveSIMM** we can: (i) prove properties of the percentile and VaR functions that hold for any permissible inputs, not just on the test data; (ii) re-validate modified models at minimal cost, reducing the need to run the new model in parallel to gain assurances about it; (iii) extract executable code directly from the verified model, eliminating differences between the specified and the production models. The scrutiny required to re-implement **openSIMM** also brought to light undocumented behaviour, namely **openSIMM**'s inability to calculate VaR figures in the upper tail of the loss distribution. Relative to testing-based approaches to model validation, formal verification provides higher assurances about model performance (for all admissible inputs, rather than just test date) and significantly reduces the costs of model maintenance.

1 Introduction

> *Testing shows the presence, not the absence of bugs.*
> *(Edsger Dijkstra [1], Turing award winner)*

Financial derivatives — defined on underlying assets such as commodities, equities, interest rate and credit instruments — commit their counterparties to future payments. In a *swap*, for example, each party commits to a stream of future payments. If either party becomes financially distressed before fulfilling its obligations, it creates *counterparty risk*.

For *cleared derivatives*, this counterparty risk is borne by a central *clearing house*, which stands by to make the requisite payments. For *non-cleared derivatives* no such third party exists, contributing to the 2007–08 financial crisis as traders withdrew from derivatives markets, unable to assess their counterparty exposure.

[*] For more information, contact Colin Rowat (CEO) at `coro@fovefi.co` or Manfred Kerber (CTO) at `make@fovefi.co`.

In 2013, the International Swaps and Derivatives Association (ISDA) responded to the Basel Committee on Banking Supervision's call for *margin requirements* — periodic payments to provide insurance against future default — by specifying the SIMM, a Standard Initial Margin Model. The SIMM is a simple Value-at-Risk (VaR) model, computing potential losses as a function of a percentile of historical prices and shocks [4]. Adopting a standard model also removed a source of potential disagreement between counterparties.

Section 2 describes openGamma's **`openSIMM`**, a "reference implementation" in Java [7]. We work with **`openSIMM`**'s original open source version; more recent versions are not open source.

Section 3 then introduces **`foveSIMM`**[2], a re-implementation of **`openSIMM`** that formally verifies specific properties that the SIMM should possess. Formal verification allows us to *prove* that desired properties hold for *all* possible inputs, rather than just *test* whether they do on a test set. From the verified **`foveSIMM`** implementation, executable Scala code[1] is automatically generated, giving assurance that the production code possesses the desired properties. For a broader discussion of formal verification in finance see [5].

2 openSIMM

In the following we summarize openGamma's **`openSIMM`**. For more detailed documentation, see [3] and [7].

`openSIMM` runs in Java 8 (and higher versions of Java), which introduces features from functional programming to Java. As such, the code can implement the SIMM's mathematical functions fairly directly. **`openSIMM`** separately computes a VaR for each of four underlying asset classes: (i) commodities (**`CO`**), (ii) equities (**`EQ`**), (iii) interest rates (**`IR`**), and (iv) credit (**`CR`**). These four values are then summed to form the portfolio VaR. Within each underlying asset class, the riskiness of assets that do not perfectly co-vary may offset each other; across underlying asset classes, they do not. We present first the general idea and then describe the specific computations.

For each underlying asset class, the associated risk is computed as a potential profit and loss (P&L) given historic data. Concretely, for each asset in the class, so-called shock vectors — lists of the same length `size` for the whole asset class containing historical changes — are read and a function is determined that computes the VaR given a confidence level.

2.1 Computing risk vectors

For each of the four asset classes, the **`shocks`** are multiplied by the number of **`units`** of each asset held and the **`rate`** (value) of a single asset. The result is then normalized with respect to a base currency by multiplying it with the exchange rate **`fxRate`**. **`openSIMM`** [7]

[1] Scala is a high-level programming language that compiles into the Java virtual machine — thus it is compatible with `openSIMM`. It combines object-oriented and functional programming. For details, see `https://www.scala-lang.org/`.

provided a simple example in which the base currency is `EUR`, and shocks are `absolute` and risks are `sensitivity` (see below for a more detailed explanation). In this case, the values are computed as the following product of three scalars (`units`, `rate`, and `fxRate`) and a vector (`shocks`):

$$\texttt{risks} = \texttt{units} \times \texttt{rate} \times \texttt{fxRate} \times \texttt{shocks} \tag{1}$$

Example 1 (Two assets and four shocks)**.** Table 1 presents a toy portfolio consisting of two `EQ` assets with four shocks each. Assume that `fxRate` = `(1/1.4) EUR`/`USD`.

assetClass	asset	currency	units	rate	fxRate	shocks
EQ	IBM	USD	100	142.63	1/1.4	[0.99, 1.02, 0.97, 1.01]
EQ	SAP	EUR	120	102.94	1/1	[1.01, 0.99, 0.97, 1.02]

Table 1: Two equities and four shocks each

The `IBM` component of the risk vector is then computed by applying equation (1):

$$\begin{aligned}\underbrace{\texttt{RiskVector}_{\texttt{IBM}}}_{[\texttt{EUR,...,EUR}]} &\approx \underbrace{\texttt{100}}_{-} \times \underbrace{\texttt{142.63}}_{\texttt{USD}} \times \underbrace{(\texttt{1/1.4})}_{\texttt{EUR/USD}} \times \underbrace{\texttt{shocks}_{\texttt{IBM}}}_{(-,\ldots,-)} \approx \texttt{10188.06} \times \texttt{shocks}_{\texttt{IBM}} \\ &\approx [\texttt{10086.18}, \texttt{10391.82}, \texttt{9882.42}, \texttt{10289.94}]\end{aligned}$$

Likewise, the `SAP` calculation yields

$$\begin{aligned}\texttt{RiskVector}_{\texttt{SAP}} &= \texttt{120} \times \texttt{102.94} \times \texttt{1} \times \texttt{shocks}_{\texttt{SAP}} = \texttt{12352.80} \times \texttt{shocks}_{\texttt{SAP}} \\ &\approx [\texttt{12476.33}, \texttt{12229.27}, \texttt{11982.22}, \texttt{12599.86}].\end{aligned}$$

The total risk vector (simulated P&L) is the sum of the two vectors (in `EUR`):

$$\begin{aligned}\texttt{risks} = \quad &[\texttt{10086.18}, \texttt{10391.82}, \ \ \texttt{9882.42}, \texttt{10289.94}] \\ + &[\texttt{12476.33}, \texttt{12229.27}, \texttt{11982.22}, \texttt{12599.86}] \\ = \quad &[\texttt{22562.51}, \texttt{22621.09}, \texttt{21864.64}, \texttt{22889.80}].\end{aligned} \tag{2}$$

The description above has a number of variants for computation, primarily depending on whether `shocks` are `relative` (to the asset's value) or `absolute`, and whether the risk type is a `sensitivity` (e.g. to a 1bp movement in an underlying interest rate) or an `exposure` (e.g. on a EUR 1mn holding) [3].

Table 2 indicates how the final `shocks` term in equation (1) is augmented in these other cases. Thus, the simplest case is that given in equation (1), above, of `absolute` shocks for `sensitivity` risks. By contrast, a `relative` shock combined with a `sensitivity` risk yields

$$\texttt{risk} = \texttt{units} \times \texttt{rate} \times \texttt{fxRate} \times (\texttt{shocks - 1}) \times (\texttt{base\_level + shift\_value}). \tag{3}$$

shock type risk type	relative	absolute
sensitivity	(shocks - 1) × (base_level + shift_value)	shocks
exposure	shocks × base_level	shocks × (base_level + shift_value)

Table 2: Computing shocks given shock and risk types

Here, base_level refers again to the current value of an underlying asset while shift_level refers to a scaling factor to avoid conditioning problems arising by division close to zero [3].

In the openSIMM repository [7] (see simm-sample) an example for shocks is provided when scenarios are missing in the test implementation; openSIMM therefore uses a utility calculation to imply them from two other scenarios, requiring more complicated FX calculations. As these calculations would not be used in production implementations (which would have all cross rates), we do not describe them.

In order to compute a VaR at a given confidence level, the percentile of the risk vector at this level is taken. The computation of the percentile is the computational core of openSIMM. We illustrate its computation with a simple example in Section 2.3.

2.2 Data files

The data are provided to openSIMM by a config file called simm.properties. This specifies base-currency, level (the VaR confidence level), and the assets with their properties (e.g. shock types as explained below) and shocks. In openSIMM's example base-currency = EUR and level = 0.9.[2] The assets, their classifications into the four risk classes, and the shocks are given to openSIMM in form of eight csv files:

1. risk-factor-definitions.csv: each underlying asset's class (CO for commodities, EQ for equities, IR for interest rates, and CR for credit), risk type (sensitivity or exposure), shock type (absolute or relative), and potentially a shift value S. For instance, with a ten day horizon, relative shocks are calculated as $\frac{p_t+S}{p_{t-10}+S}$, and absolute shocks as $p_t - p_{t-10}$, where p is a historical return for a given risk factor.

2. risk-factor-base-levels.csv: the base levels of the assets (that is, the current values of the assets in whatever units the assets are quoted in);

3. risk-factor-shocks.csv: the shocks (whether relative or absolute) for the risk factors;

4. fx-rates.csv: the current fxRates between the currencies;

5. fx-rate-shocks.csv: shocks to the fxRates;

[2] The variable is called level in the openSIMM Java files, but var-level in the simm.properties file.

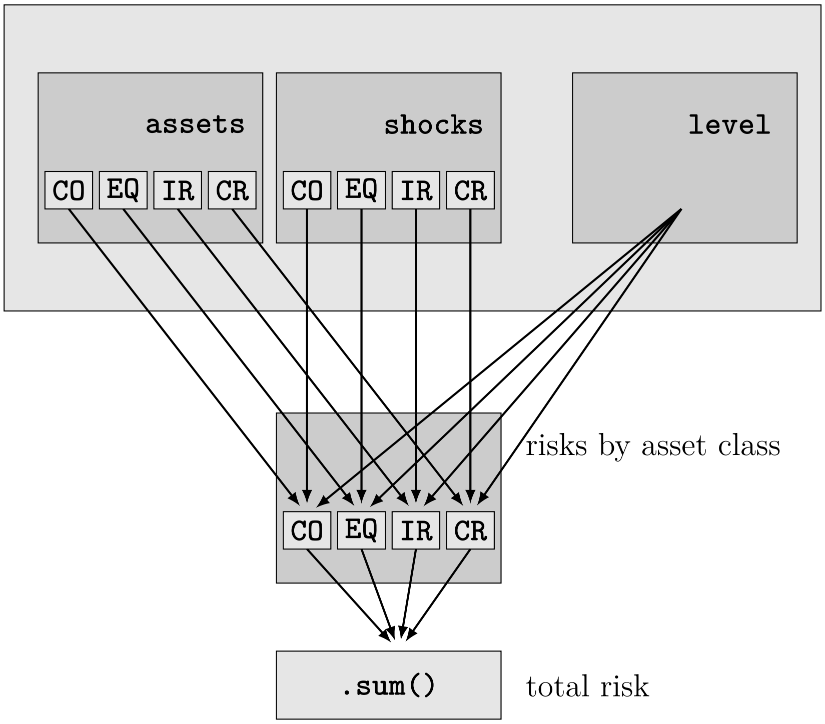


Figure 1: Risks in underlying asset classes computed as percentiles.

6. **portfolio-derivatives.csv**: the holdings of each asset and the currencies in which they are held;

7. **portfolio-initial-margin.csv**: the initial margin of the assets;

8. **portfolio-variation-margin.csv**: the portfolio variation margins of assets (adjustements made to the intial margin, usually daily, in response to changing market prices).

Schematically the total risk is computed as the sum of the risks in the four underlying asset classes as indicated in Figure 1. The input consists of the portfolio, the historic shocks, and the risk level (as explained in Section 2.3). The program was specified and then written by human programmers as seen in Figure 2. Typically programming follows the specification, but has to be more detailed. Whether the program satisfies the specification is usually determined by testing against test cases but is not established in general.

2.3 Computing the percentile

This section describes **openSIMM**'s computation of the percentile, which is done in lines 32–47 of the **SimmUtils.java** Java method **percentile**. The code, displayed in Code

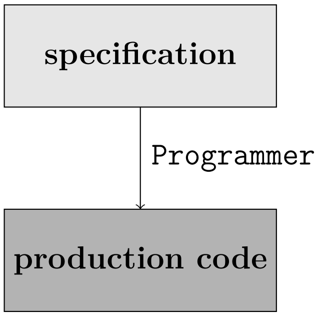


Figure 2: Traditional programming approach: from specification to code.

Fragment 1 (on p. 8), takes in a list of values (used for the P&L values) and computes the percentile at a certain `level`. We illustrate this by means of a toy example.

Given simulated P&L values for an underlying asset class,

$$\texttt{values} = \texttt{[8, 10, 1, 4, 5]}.$$

whose length defines a `size` variable. This is implemented in line 34 of the code fragment; in the present example, it takes on a value of `size` $= 5$.

To compute a particular percentile, here 0.75, line 35 of the code fragment first ensures that it is in scope, throwing an exception if the percentile sought is above `1` − `(0.5 / size)` = `0.9`. Otherwise, the following steps are taken:

1. Sort the list, yielding `sorted` = `[1, 4, 5, 8, 10]`. Lines 37–39 of the code fragment implement this step.

2. Map the elements of the sorted list to elements on the unit interval `[0;1]` at equal distances of $\frac{\texttt{1}}{\texttt{size}} = \frac{\texttt{1}}{\texttt{5}}$ with `size` = `size(values)`, positioning them so that the values stay half of the distance from the ends of the unit interval, that is the smallest value being $\texttt{0 +}\ \frac{\texttt{1}}{\texttt{2}\times\texttt{size}}$ and the biggest as $\texttt{1 -}\ \frac{\texttt{1}}{\texttt{2}\times\texttt{size}}$. We describe this by a function `equidistant_points_on_unit_interval_of`. This function is not explicitly defined in `openSIMM`, but implicitly used in lines 41–43 of Code Fragment 1. `equidistant_points_on_unit_interval_of` = `[0.1, 0.3, 0.5, 0.7, 0.9]`.

3. For input values (of the percentile function) in the so defined range (that is, the five values `0.1`, `0.3`, `0.5`, `0.7`, and `0.9` in the example), the percentile is then the corresponding value in the sorted list. Thus, for example, the `0.3` percentile is `4`, as shown in Table 3.

`level`	`0.1`	`0.3`	`0.5`	`0.7`	`0.9`
`percentile(level)`	`1`	`4`	`5`	`8`	`10`

Table 3: Percentiles corresponding to a list of five numbers

4. For values between those, that is, for **`level`** in an open interval (**`lower`**, **`upper`**), linearly interpolate:

$$\texttt{percentile(level)} = \texttt{lowerValue} + (\texttt{level} - \texttt{lower}) \times \frac{\texttt{upperValue} - \texttt{lowerValue}}{\texttt{upper} - \texttt{lower}} \tag{4}$$

   where **`lower`** and **`upper`** are adjacent points in the equidistant points and **`lowerValue`** and **`upperValue`** are the corresponding values in **`sorted`**. In the code fragment, lines 44-45 determine **`lowerValue`** and **`upperValue`**; line 46 returns the result of applying equation (4).

   To compute, for instance, the percentile at **`0.75`**, apply equation (4) with **`lower`** = **`0.7`**, **`upper`** = **`0.9`** and the corresponding values in **`sorted`** are **`lowerValue`** = **`8`** and **`upperValue`** = **`10`**, for:
   `8 + (0.75 - 0.7)` × `(10 - 8)/(0.9 - 0.7) = 8.5`.

Figure 3 displays the result of this construction, which is defined over the closed interval:

$$\texttt{[1/(2} \times \texttt{size), 1 - 1/(2} \times \texttt{size)]}$$

a strict subset of the unit interval.

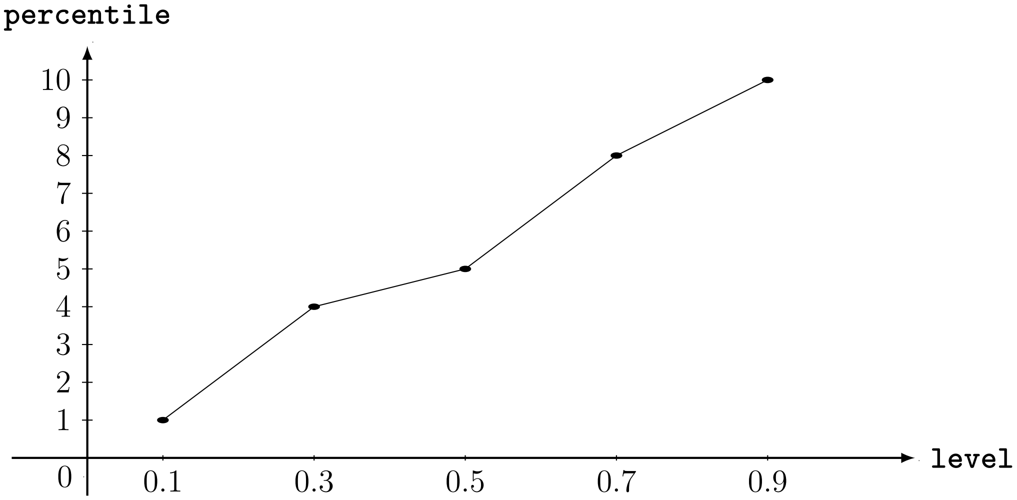


Figure 3: Constructing a percentile function from **`values = [8, 10, 1, 4, 5]`**

To summarize, the **`percentile`** function is the **`linear_interpolation`**[3] of the partial function given by the mapping from **`equidistant_points_on_unit_interval`** to the sorted **`values`** vector.[4] The function is applied to a particular **`level`** in the interval

[3] As with **`equidistant_points_on_unit_interval`**, there is no explicit mention of **`linear_interpolation`** in **`openSIMM`**, but it is computed in lines 41–46 of Code Fragment 1.

[4] Partial functions are functions that are not defined everywhere. For example, $y = \frac{1}{x}$ is undefined at $x = 0$, while $y = \log x$ is undefined for $x \leq 0$.

[0.5/size(values), 1-0.5/size(values)]. In the example above, the percentile function is not defined below **0.1** and above **0.9**: this is a limitation of the 2016 open-source implementation of **openSIMM**.

```
public static double percentile(List<Double> values, double level) {

  int size = values.size();
  ArgChecker.isTrue(level < 1.0d - 0.5d / size, "level not within the data range");

  List<Double> sorted = values.stream()
      .sorted(Double::compare)
      .collect(toList());

  int i = (int) Math.ceil(size * level - 0.5);
  double lower = (i - 0.5) / size;
  double upper = (i + 0.5) / size;
  double lowerValue = sorted.get(i - 1);
  double upperValue = sorted.get(i);
  return lowerValue + (level - lower) * (upperValue - lowerValue) / (upper - lower);
}
```

Code Fragment 1: **openSIMM** Java implementation of the **percentile** method

3 foveSIMM

In this section we first describe how **foveSIMM** re-implements **openSIMM**. Then we present some experiments with code modification.

3.1 foveSIMM implementation

foveSIMM re-implements **openSIMM** and formally verifies properties of the percentile function. It does so in Isabelle, a powerful proof assistant [6] from which executable Scala code is then automatically generated.

We shall describe two parallel implementations of the percentile function:

1. **percentile** captures the abstract properties of a percentile function — in our case, linear interpolation — so is well suited to proving properties;

2. **percentile_impl** (for *implementation*) captures the computational aspects of a percentile function, easing the extraction of — in our case, executable Scala code.

We formally prove the equivalence of **percentile** and **percentile_impl** by a so-called bridging theorem.

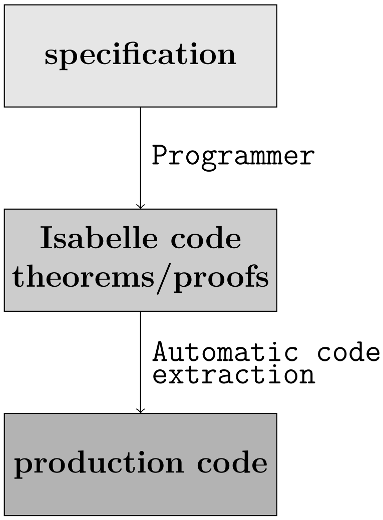


Figure 4: Proof-based programming approach in **foveSIMM**: from specification through definitions and theorems to verified executable code.

percentile, the abstract definition, is formed by the same procedure as described above for **openSIMM**: a partial function associates each point in a sorted list of shocks with equidistant points on the unit interval, leaving half the interval length at each end; the full function, **percentile**, then linearly interpolates between the points of the partial function. Formally in Isabelle this is:

```
definition percentile :: "real list ⇒  real ⇒  real" where
  "percentile ys x = linear_interpolation
             (equidistant_points_on_unit_interval_of (sort ys)) x"
```

Code Fragment 2: Abstract Isabelle definition of the percentile function

percentile_impl, the computational version of the percentile function in Code Fragment 3, is very close to the **openSIMM** Java method Code Fragment 1 and can — unlike the definition in Code Fragment 2 — be evaluated with concrete values in Isabelle. It also can be automatically extracted as a corresponding Scala method.

```
definition percentile_impl :: "real list ⇒  real ⇒  real"
where
 "percentile_impl values level =
    (let  (size :: real) = (real (length values));
           (sorted :: (real list)) = sort values;
           i = (ceiling (size * level - 0.5));
           (lower :: real) = (i - 0.5) / size;
           (upper ::real) = (i + 0.5) / size;
           (lower_value :: real)  = sorted ! (nat (i-1));
           (upper_value :: real)  = sorted ! (nat i)
       in lower_value +
       (level - lower) * (upper_value - lower_value) / (upper - lower))"
```

Code Fragment 3: Computational Isabelle definition of the percentile function

With the Isabelle formalizations, we formally prove four properties of the linear interpolation in equation (4).[5] As the percentile is built on linear interpolation, these properties imply properties of the percentile function as well.

First, a linear interpolation has a lower and an upper bound:

Lemma 3.1. *For a partial function p with finite domain, the linear interpolation is bounded from above by the maximum of the values of p, and from below by the minimum.*

Formally, the corresponding lemma for the upper bound is written in Isabelle as (with **ran** and **dom** standing for "range" and "domain", respectively):

```
lemma upper_bound:
  assumes "finite (dom p)"
  assumes "x ∈ defined_interval_of p"
  shows "linear_interpolation p x ≤ Max (ran p)"
```

The definitions and lemmas are written by the human user. Proofs are developed by the human user with system support and checked for correctness by the Isabelle system

Second, the linear interpolation of a finite set of weakly increasing numbers is weakly monotonously increasing.

Lemma 3.2. *For a partial function p with finite domain so that for all x and x' for which p is defined with $x \leq x'$: $p(x) \leq p(x')$ holds if $x \leq x'$ then $linear\_interpolation(p, x) \leq linear\_interpolation(p, x')$.*

Formally, the corresponding lemma is in Isabelle:

```
lemma linear_interpolation_monotonic:
  assumes "finite (dom p)"
  assumes "x ∈ defined_interval_of p" "x’ ∈ defined_interval_of p"
  assumes "∀x x’ y y’. x ≤ x’ ∧ p x = Some y ∧ p x’ = Some y’ ⟶ y ≤ y’"
  assumes "x ≤ x’"
  shows "linear_interpolation p x ≤ linear_interpolation p x’"
```

Third, the linear interpolation is linear:

Lemma 3.3. *For a partial function p with finite domain and any real number, c, $linear\_interpolation(p_c, x) = c \cdot linear\_interpolation(p, x)$, where p_c is the partial function given by multiplying the results of p — if defined — by c.*

[5] The proofs can be found in the Isabelle code on `https://github.com/fovefi/foveSIMM`.

Formally, the corresponding lemma is expressed in Isabelle:

```
  lemma linear_interpolation_scale:
  assumes "finite (dom p)"
  assumes "x ∈ defined_interval_of p"
  shows "linear_interpolation (λx. map_option (λy. c * y) (p x)) x =
            c * linear_interpolation p x"
```

Fourth, the linear interpolation is Lipschitz continuous, so the difference of the function values can be bounded by the difference times a Lipschitz constant:

Lemma 3.4 (Lipschitz continuity)**.** *For a partial function p with finite domain there exists a Lipschitz constant K such that for all x and x'*

$$|\mathit{linear\_interpolation}(p, x') - \mathit{linear\_interpolation}(p, x)| \leq K \cdot |x' - x|.$$

Formally, the corresponding lemma in Isabelle is:

```
lemma linear_interpolation_Lipschitz:
  assumes "finite (dom p)"
  assumes "x ∈ defined_interval_of p" "x' ∈ defined_interval_of p"
  shows "∃K. abs (linear_interpolation p x' - linear_interpolation p x)
             ≤ K * abs (x' - x)"
```

As **foveSIMM**'s percentile function is formed by linear interpolation, the four properties of linear interpolation proved above can also easily be proved to hold for **foveSIMM**'s **percentile** function (q.v. Code Fragment 2). Consequentially, **foveSIMM** (i) bounds VaR for *any* portfolio; (ii) ensures that VaR are defined for *any* confidence level of the percentile (in the range given by **openSIMM**); (iii) ensures that VaR scales with portfolio size for *any* portfolio; and (iv) given VaR at some confidence level, bounds it at *any* other.

The bridging theorem stating the equivalence between the **percentile** function and the **percentile_impl** function is stated and proved in Isabelle as follows:

```
theorem percentile_java_equiv:
  assumes "values ≠ []"
  assumes "1 / real (2 * length values) ≤ level"
  assumes "level ≤ 1 - 1 / real (2 * length values)"
  shows "Percentile.percentile values level = percentile_impl values level"
  using assms less_eq_real_def percentile_java_equiv_except_left
        percentile_java_equiv_left by blast
```

To gain further confidence in **foveSIMM**, it may be validated against **openSIMM**. Table 4 shows that **foveSIMM** reproduces the VaR numbers from **openSIMM**'s reference example.[6]

[6] The values are provided in CSV files on `https://github.com/OpenGamma/OpenSIMM/tree/master/src/test/resources/simm-sample`. **openSIMM**'s computation then takes place in `https://github.com/OpenGamma/OpenSIMM/blob/master/src/test/java/com/opengamma/opensimm/SimmTest.java`. These computations use the FX calculations mentioned above, which we have not described. Agreement between the two implementations still leaves open the validity of the **openSIMM** calculations.

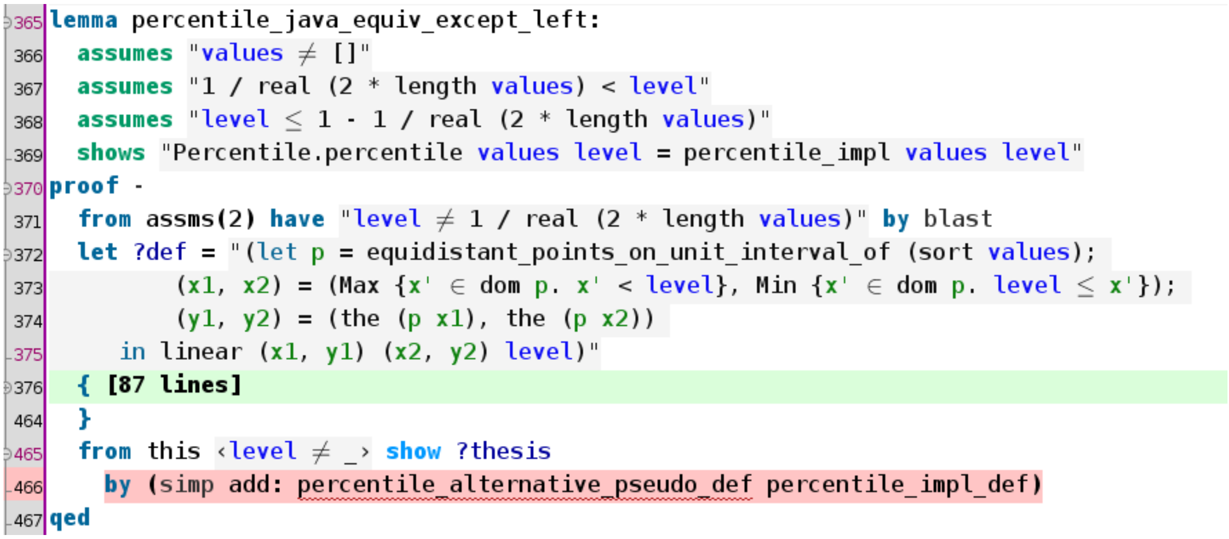
Asset Class	openSIMM result	foveSIMM result	foveSIMM decimal
CO	564.3703	493824 / 875	564.3703
EQ	740.7143	5185 / 7	740.7143
IR	447.5351	12493389 / 27916	447.5351
CR	33.3300	3333 / 100	33.3300
total_risk	1785.9496	222573974 / 124625	1785.9496

Table 4: Comparison of **openSIMM** and **foveSIMM** computations

Any changes in a definition may break the proof of some properties. For instance, let us change the definition of **percentile** in Code Fragment 2 by 'forgetting' to sort the list **ys** first:

```
"percentile ys x = linear_interpolation
   (equidistant_points_on_unit_interval_of ~~sort~~ ys) x"
```

Now proofs such as that of a lemma used to prove the equivalence of the abstract **percentile** and computational **percentile_impl** functions fail (see Figure 5).



```
lemma percentile_java_equiv_except_left:
  assumes "values ≠ []"
  assumes "1 / real (2 * length values) < level"
  assumes "level ≤ 1 - 1 / real (2 * length values)"
  shows "Percentile.percentile values level = percentile_impl values level"
proof -
  from assms(2) have "level ≠ 1 / real (2 * length values)" by blast
  let ?def = "(let p = equidistant_points_on_unit_interval_of (sort values);
          (x1, x2) = (Max {x' ∈ dom p. x' < level}, Min {x' ∈ dom p. level ≤ x'});
          (y1, y2) = (the (p x1), the (p x2))
     in linear (x1, y1) (x2, y2) level)"
  { [87 lines]
  }
  from this ‹level ≠ _› show ?thesis
    by (simp add: percentile_alternative_pseudo_def percentile_impl_def)
qed
```

Figure 5: The proof of a lemma used in the proof of the bridging theorem fails as indicated indicated by the red line under the lemma's name.

If the proofs cannot be patched and the properties are known consequences of the definitions, then some definition is necessarily incorrect. Thus, successful proofs for known results give confidence in the correctness of the definitions.

However, successful proofs can also mask incorrect definitions.[7] For example, formally

[7] Wos and Winker used an autmated theorem prover to derive results for finite semigroups from an incorrect definition of 'involution' [8]. As they knew little about finite semigroups, they only discovered the mistake by speaking to an expert.

verifying the properties above does not prove that **percentile** is actually a percentile function. For example, we can mis-define the percentile function in a way that the theorems still hold. We modified the definitions of **percentile** and **percentile_impl** by shifting them by ε to the left, so that

```
percentile_new ys x = percentile ys (x+ε);
```

as illustrated in Figure 6. Although this broke proofs above, it took little effort to restore them. With $\varepsilon = 0.01$, the modified code produced a total risk with an error of 4.5%. With $\varepsilon = 0.1$ the computed risk has an error of nearly 20%.

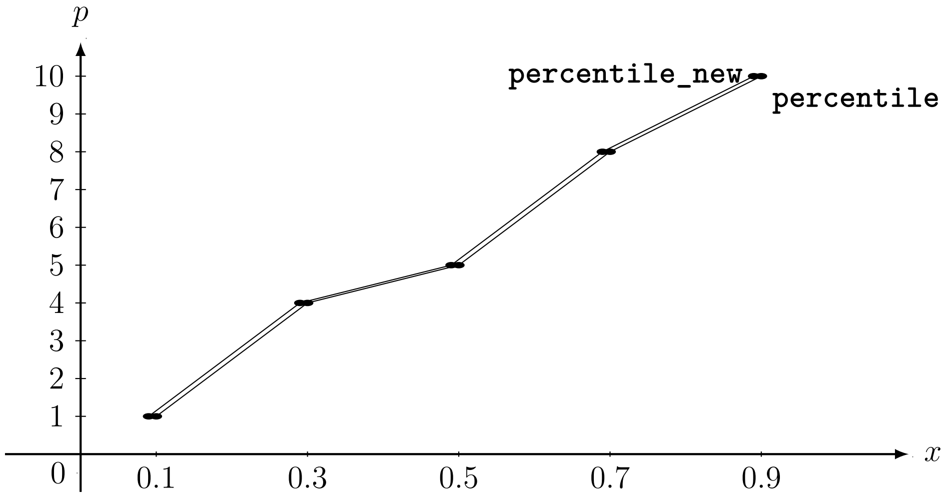


Figure 6: The shifted **percentile_new** function.

Thus, formal verification still requires human validation: humans must still check that the definitions provided to the theorem prover are correct and scrutinize the theorem statements, and must still determine what theorems are required. However, the human user does not need to check the correctness of the proofs, nor the correctness of the automatically extracted production code.

3.2 Re-validating modified code

To assess how easy it is to re-validate modified code, we performed two experiments.

The first starts by observing that **openSIMM**'s Java implementation does not define the percentile function on the full interval $[0, 1]$: for values above **1 - (0.5 / size)**, an exception is thrown (line 35 in Code Fragment 1), and no risk value is computed. This results from the percentile function's definition as an interpolation, with no extrapolation beyond the largest point in **values**. In the example in Section 2.3, the VaR cannot be computed above the 90% percentile. This is unsatisfactory as risk management typically focuses on the upper tail.

To remedy this, we linearly extrapolated above `1- (0.5 / size)` and — less interestingly — below the lower boundary at `0.5 / size` to create a percentile function defined on the whole unit interval $[0, 1]$. Figure 7 illustrates the extrapolation, in which the dashed line extends the neighbouring interpolated slope.

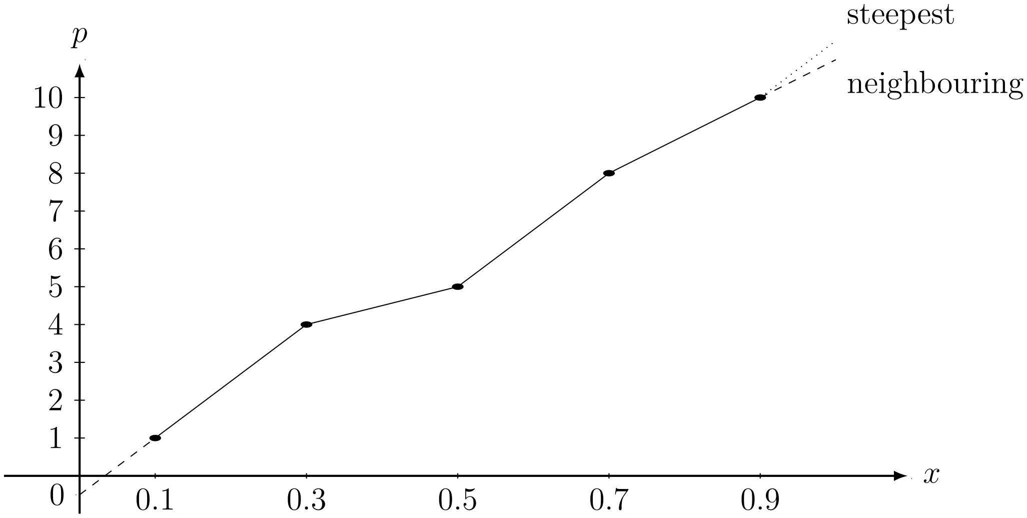


Figure 7: Linearly extrapolating the percentile function to the full unit interval, $[0, 1]$

Re-validation of the modified model requires two further changes. First, the statements of some lemmas change. For example, the statement of boundedness Lemma 3.1 must be modified to recognize that the percentile function is no longer bounded (from above) by the maximal element in **`values`** but by the extrapolation at value 1. Second, some proofs must be modified, even if the lemmas are still true: specifically, proofs must cover the newly extrapolated parts of the function, rather than just the original interpolated parts.

This experiment took about a day, compared to circa two weeks for the original development.

The second experiment imagined a validator or regulator who was concerned that the extrapolation in the first experiment was overly optimistic: they want the extrapolation to be based on the steepest slope in the original percentile function rather than on the neighbouring one. In the example plotted in Figure 7, this only affects the right tail (plotted with the dotted line). Now only the definition of the extrapolation and some proofs needed adaptation — some three hours' work. Since the properties are formally verified, no practical experiments — which may take weeks or months — are necessary for re-validation.

4 Conclusions

We estimate that reimplementing **openSIMM** in a formal theorem proving environment takes a similar amount of time as implementation in a language like Java takes. However, doing so allows stronger assurances about the model's performance. We do not attempt to compare the validation and verification costs associated with, for example, testing **openSIMM** to the higher levels of assurance associated with **foveSIMM**'s formal verification. However, the costs of re-validating modified models is significantly reduced by formal verification. Once formal methods become more widely used and earn the trust of regulators, the approval process for regulated models can also be streamlined.

References

1. Dijkstra, E. W. In: Software Engineering Techniques — NATO Science Committee. Oct. 1969, p. 16. URL: `http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1969.PDF`.
2. *fovefi/foveSIMM: formally verified re-implementation of openGamma's openSIMM (Standard Initial Margin Model).* May 2019. URL: `https://github.com/fovefi/foveSIMM`.
3. Henrard, M. *SIMM: an HVaR approach implementation.* Margining Documentation 6. London: OpenGamma, Mar. 2015.
4. ISDA. *Standard Initial Margin Model for Non-Cleared Derivatives.* type. International Swaps and Derivatives Association, Dec. 2013. URL: `https://www.isda.org/a/cgDDE/simm-for-non-cleared-20131210.pdf`.
5. Kerber, M., C. Rowat, and N. Vosloo. "Using formal verification to develop higher assurance, more maintainable financial software". *Journal of Risk Management in Financial Institutions* 13.1 (Jan. 2020).
6. Nipkow, T., L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic.* Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2003. URL: `https://books.google.co.uk/books?id=xwdqCQAAQBAJ`.
7. *OpenGamma/OpenSIMM: reference implementation of the ISDA-proposed Standard Initial Margin Model (SIMM) for non-cleared derivatives.* May 2019. URL: `https://github.com/OpenGamma/OpenSIMM`.
8. Wos, L. and S. Winker. "Open questions solved with the assitance of AURA". In: *Automated theorem proving: after 25 years.* Ed. by W. Bledsoe and D. Loveland. Contemporary Mathematics 29. American Mathematical Society, 1984, pp. 73–88.