Itovsn3 in *AXIOM*: modules, algebras and stochastic differentials

by

W.S. Kendall

June 1999

This is `/home/bach/wsk/ms/Axiom/itovsn3.w` (`unix`) version: 2.18.
It was last edited on 10:44:17, 14/06/1999.

Itovsn3 in *AXIOM*: modules, algebras and stochastic differentials

Wilfrid S. Kendall,
Statistics,
University of Warwick
Coventry CV4 7AL, UK

Abstract

This document describes version 2.18 of an actively developing implementation of *Itovsn3* using the special object-oriented and mathematical-programming features of *AXIOM* code. It supersedes Version 1, which was a direct translation from (a simplified form of) the *Mathematica* version of [9], which itself was derived from the *REDUCE* version described in [6] and at greater length in [5]. Examples are given of applications of *Itovsn3* to Bessel processes, to Brownian motion on the n-sphere, and to derivations of the Black-Scholes formula and the Mardia-Dryden shape density.

Contents

Identifiers used in the code

Introduction

For some time computer algebra has been applied in statistics and in probability [8, 10], with particular application to Itô calculus [5, 6]. It is the purpose of this work, supported by EPSRC grant GR/K71677, to exploit the features of the innovative computer algebra package *AXIOM* in a new implementation of Itô calculus. The task of this research report is to record the current implementation and partial testing of the various domains and packages which comprise *Itovsn3*.

Version 1 was a direct translation of the *Mathematica* version of *Itovsn3*, reported on in an earlier version of this document. As an implementation it was essentially complete but did not use any of the information-hiding facilities of *AXIOM*. Version 2 is described by this report[1], and does use *AXIOM* information-hiding *via* packaging and type facilities. Version 3 will be the production version (to conform to *Itovsn3* version numbering for other computer algebra packages!)

This document uses Briggs' `Nuweb` program [2], a small, free and portable `C` program based on Knuth's idea of *literate programming* [12, 14]. This enables the production of the LaTeX source `itovsn3.tex` and the various source code and test files from a single `Nuweb` file `itovsn3.w`, using the invocation

```
nuweb itovsn3
```

(note that the `Nuweb` output may signal that one needs to repeat this after running LaTeX on `itovsn3`, so as to sort out the labelling of the various scraps of code). The insertions of numbered scraps of computer code at various places in the text of this report comprise the actual source code for *Itovsn3* and various test files - what you see is what `Nuweb` makes sure the computer gets. The relevant file (suffix `.spad` for source file, `.input` for test file) is to be found at the head of the scrap. An index of identifiers is given at the start of the document.

Three scraps at the start of this section are exceptions to the above rule: `revision and author information`, `synonyms`, `silence please` are *macros*, referred to by other scraps throughout the document to abbreviate repetitive code of a housekeeping nature. The first macro is placed in every *AXIOM* file produced by the `Nuweb` source of this report. It identifies the source of the *AXIOM* code and its revision and author information, including the date of last modification:

[1] This is `/home/bach/wsk/ms/Axiom/itovsn3.w` (`unix`) version: 2.18.
It was last edited on 10:44:17, 14/06/1999.

⟨revision and author information 2a⟩ ≡
```
-- Code derived from $Source: /home/bach/wsk/ms/Axiom/itovsn3.w,v $
--                   $Revision: 2.18 $ $State: U $
--                   $Date: 1999/06/14 10:44:17 $
-- Author: Wilfrid S. Kendall
```
◇

Macro referenced in scraps 2d, 7, 11a, 13, 19b, 23b, 27, 32b, 33d, 37, 40a, 45a, 49a, 50c, 53a, 59.

The second macro sets up some useful *synonyms*.

⟨synonyms 2b⟩ ≡
```
)synonym bye quit
)set quit unprotected
)synonym ls system ls
)synonym tex set output tex
```
◇

Macro referenced in scraps 2cd.

The synonym `tex` allows us to switch to and from LaTeX output using

```
)tex on
 ... results are output first in standard form then in LaTeX form
)tex off
```

The third and final macro `silence please` is used for test input files. It clears the workspace, sets up silent running (so as not to obscure the important output!) and adds the useful synonyms noted above.

⟨silence please 2c⟩ ≡
```
)clear completely
)set messages autoload off
-- Very verbose output would hide whether
-- the code is doing its job!
```
⟨synonyms 2b⟩
◇

Macro referenced in scraps 11a, 23b, 32b, 33d, 40a, 45a, 50c, 53a.

The current version of the package is compiled and loaded by the invocation `)read itovsn3` from within *AXIOM*. This reads in the file `itovsn3.input`, which contains

1. revision and author information and synonyms;

   `"itovsn3.input"` 2d ≡
   ```
   -- itovsn3.input
   ```
   ⟨revision and author information 2a⟩
   ⟨synonyms 2b⟩
   ◇

   File defined by scraps 2d, 3.

2. instructions to compile the different domains and packages;

```
"itovsn3.input" 3 ≡
      )compile bsd
      )compile sd
      )compile sd2
      )compile ito
      )compile ito2
      )compile itointegral
      )compile oldito
      ◇
```

File defined by scraps 2d, 3.

Thus the *AXIOM* version of *Itovsn3* is produced from the `Nuweb` source `itovsn3.w` by the following sequence of commands:

```
nuweb itovsn3
```

(then enter *AXIOM*)

```
)read itovsn3
```

(then either exit *AXIOM* or conduct an *AXIOM* session).

After compilation has been successfully completed, it suffices to load the various modules using the `)library` directive. We do not actually require `oldito` except for some test purposes, so the following will work well:

```
)library BSD SD SD2 ITO ITO2 ITOINT
```

Chapter 7 describes a session of *Itovsn3* which may be undertaken once this initial compilation phase has been completed: this is contained in the *AXIOM* script `MardiaDryden.input`. Other test scripts are produced in the course of invoking `Nuweb`. We conclude this introduction with a list of these scripts.

bsd.input tests the `BSD` module, which implements basic stochastic differentials;

sd.input tests the `SD` module, which provides modules of stochastic differentials built on basic stochastic differentials, and also the helper package `SD2`, which ensures that the `EXPR(INT)` implementation of `SD` operations is used by default;

ito.input tests the Itô differential, implemented in the package `ITO`.

bessel.input demonstrates the application of `ITO` for Bessel processes, and also checks out the general Itô formula for two variables;

ito2.input tests associated procedures, provided in the package `ITO2`;

sphere.input demonstrates the use of the procedures in `ITO2` by investigating Brownian motion on the n-sphere;

itointegral.input tests simple Itô integral procedures.

Acknowledgements

This research was supported by the EPSRC under grant GR/K71677.

Chapter 1

The mathematical setting

We begin by describing the underlying mathematical setting. We start here because *AXIOM*'s "mathematical" style of programming is best initiated by considering the tasks we wish the software to perform, but expressed in *mathematical* terms. The algorithmic basis for formal manipulations in the semimartingale stochastic calculus, and indeed for previous implementations of *Itovsn3*, is Itô's theory of modules of stochastic differentials [4]. Thus the principal implementation task is to define domains and packages of computation to correspond to:

- the module $\mathcal{D}_2$ of stochastic differentials representing a given filtration (actually $\mathcal{D}_2$ is an *algebra* over the function algebra $\mathcal{P}$ of bounded previsible random functions, as it has a multiplication);

- the Itô differential operator `d`, which converts an expression (assumed to define a semimartingale) into a stochastic differential;

- the operation of taking the drift `drift` $\mathtt{d}\,x$ of a stochastic differential $\mathtt{d}\,x$, a module-linear operation mapping $\mathcal{D}_2$ into a subset $\mathcal{D}_1$ of classical differentials.

Various operations tie this all together. Setting aside the usual object-oriented functions such as constructors and destructors, we require:

- a list of *basic* stochastic differentials and corresponding semimartingales;

- implementation of the $\mathcal{P}$-linearity of $\mathcal{D}_2$;

- the *stochastic differential* $\mathtt{d} : \mathcal{S} \to \mathcal{D}_2$, which maps basic semimartingales to corresponding stochastic differentials and obeys *Itô's formula*;

- the *bracket multiplication* $* : \mathcal{D}_2 \times \mathcal{D}_2 \to \mathcal{D}_2$, which is $\mathcal{P}$-module-linear, and actually maps $\mathcal{D}_2 \times \mathcal{D}_2$ into its annihilator $\mathcal{D}_1$ under `*` (corresponding to the $\mathcal{P}$-module of *classical* differentials);

- the *drift* `drift` $: \mathcal{D}_2 \to \mathcal{D}_1$, which is $\mathcal{P}$-module-linear, and projects onto $\mathcal{D}_1$;

- the *stochastic integral* `itoIntegral` : $\mathcal{D}_2 \to \mathcal{S}_0$, which is a left-inverse to `d` in the sense that

$$\mathtt{d} \circ \mathtt{itoIntegral} \quad \equiv \quad \mathbb{I} \qquad \text{on } \mathcal{D}_2 \,.$$

Notice here the *AXIOM* convention, to which *Itovsn3* adheres, that multi-part names are signalled by capitalization of second and further components (hence, `itoIntegral`), but *not* the first component (hence, `drift`).

For the present we do not consider implementation of the algebra $\mathcal{P}$ of bounded previsible random functions, but simply view all `Expression` quantities as semimartingales, whose randomness is defined by the way in which they use basic semimartingales.

Finally, we choose not to implement the distinction between general stochastic differentials (elements of $\mathcal{D}_2$) and "classical" differentials with no quadratic variation (elements of $\mathcal{D}_1 \subset \mathcal{D}_2$). This is mainly for the sake of simplicity. However note that if we did do this then a general semimartingale differential could belong to $\mathcal{D}_2 \setminus \mathcal{D}_1$ in general but might belong to $\mathcal{D}_1$ when restricted to a suitable timeset. Such a distinction may be implemented in a future revision, defining `ClassicalDifferential` as a *subdomain* of `StochasticDifferential`, but then due caution will have to be exercised, as illustrated in the following example:

Example 1.1 : Suppose B is a standard Brownian motion. Now consider the differential

$$\mathbb{I}_{[t>1]}\, \mathtt{d}B \;+\; \mathbb{I}_{[t\leq 1]}\, \mathtt{d}t \,. \tag{1.1}$$

This is classical over the time interval $[0,1]$ and non-classical over the time interval $(0,\infty)$. In general it seems infeasibly burdensome (in computational terms) to have to decide automatically over which regions the stochastic differential was in fact classical.

Chapter 2

The domain of basic stochastic differentials

The first step is to define a domain of *basic stochastic differentials*, which defines the symbols representing those elementary stochastic differentials which are used to generate the module $\mathcal{D}_2$. We base it on the original *AXIOM* domain of `Symbol` with an important difference: in order to convert from `String` (or indeed `Symbol`) to `BasicStochasticDifferential` (abbreviated `BSD`) one must first have declared the relevant variable `dX` as a `BSD` (obtained from `X` as its stochastic differential) using the function `introduce!(X,dX)`. (This restriction is deliberate: the alteration of symbols denoting members of `BSD` corresponds in stochastic calculus terms to a rather drastic procedure affecting the filtration!)

We begin with some special comments (reproduced automatically in the online documentation of *AXIOM* after this code has been compiled) and a line `BasicStochasticDifferential` ... which determines the fundamental algebraic nature of the `BSD` domain.

"bsd.spad" 7 ≡

```
-- bsd.spad
⟨revision and author information 2a⟩
)abbrev domain BSD BasicStochasticDifferential
--% BasicStochasticDifferential
-- Compilation order:
-- BSD SD SD2 ITO ITO2 ITOINT OLDITO
-- ***
++ Basic Operations: introduce!, copyBSD, copyIto, getSmgl
++ Related Domains: StochasticDifferential(R)
++ Also See:
++ AMS Classifications:
++ Keywords: stochastic differential, semimartingale.
++ Examples:
++ References:
++ Ito (1975), Kendall (1991a,b; 1993a,b; 1999a,b).
++ Description:
++ Based on \axiom{Symbol}: a domain of symbols
++ representing basic stochastic differentials, used
++ in \axiom{StochasticDifferential(R)} in the underlying
++ sparse multivariate polynomial representation.
```

```
++
++ We create new \axiom{BSD} only by coercion from \axiom{Symbol}
++ using a special function \axiom{introduce!} first of all to
++ add to a private set \axiom{SDset}. We allow a separate
++ function \axiom{convertIfCan} which will check whether the
++ argument has previously been declared as a \axiom{BSD}.
BasicStochasticDifferential(): Category == Implementation where
 INT ==> Integer
 OF  ==> OutputForm
 Category ==> OrderedSet with
  ConvertibleTo(Symbol)
```
◇

File defined by scraps 7, 8ab, 9abc, 10abc.

Note that we are using *AXIOM*'s "pile" convention, which defines blocks by indentation.

We require the following:

1. Conversion to `Symbol`. This is derived from the category declaration: a `BSD` object can be converted to a `Symbol` but only by an explicit conversion, not automatically by the compiler. The conversion from `Symbol` is possible but can fail. It is better programming style in *Itovsn3* to link such conversions closely to invocations of `introduce!`:

   `"bsd.spad"` 8a ≡

   ```
   convertIfCan: Symbol -> Union(%, "failed")
    ++ convertIfCan(ds) transforms \axiom{dX} into a \axiom{BSD}
    ++ if possible (if \axiom{introduce(X,dX)} has
    ++ been invoked previously).

   convert: Symbol -> %
    ++ convert(dX) transforms \axiom{dX} into a \axiom{BSD}
    ++ if possible and otherwise produces an error.
   ```
   ◇

   File defined by scraps 7, 8ab, 9abc, 10abc.

2. Creation of new basic stochastic differentials derived from semimartingales (the notion of a semimartingale is left *implicit*, as at present we do not have a type for `Semimartingales`, so in *AXIOM* a semimartingale has the uninformative type of `Symbol`);

   `"bsd.spad"` 8b ≡

   ```
   introduce!: (Symbol,Symbol) -> Union(%, "failed")
    ++ introduce!(X,dX) returns \axiom{dX} as \axiom{BSD} if it
    ++ isn't already in \axiom{BSD}
   ```
   ◇

   File defined by scraps 7, 8ab, 9abc, 10abc.

   As noted above, we do *not* allow implicit coercion to define a member of `BSD` which has not been formally introduced to the package.

"bsd.spad" 9a ≡

3.
```
        d: Symbol -> Union(%,INT)
         ++ d(X) returns \axiom{dX} if \axiom{tableIto(X)=dX}
         ++ and otherwise returns \axiom{0}
```
◇

File defined by scraps 7, 8ab, 9abc, 10abc.

In a more developed implementation we would define `d` *via* a domain of `Semimartingales`.

4. Display of private sets `setBSD` (of basic stochastic differentials), `tableIto` (of pairs of semimartingales and basic stochastic differentials), and `getSmgl` to extract the basic semimartingale corresponding to a basic stochastic differential:

"bsd.spad" 9b ≡

```
        copyBSD:() -> List %
         ++ copyBSD() returns \axiom{setBSD} as a list of \axiom{BSD}.
        copyIto:() -> Table(Symbol,%)
         ++ copyIto() returns the table relating semimartingales
         ++ to basic stochastic differentials.
        getSmgl: % -> Union(Symbol,"failed")
         ++ getSmgl(bsd) returns the semimartingale \axiom{S} related
         ++ to the basic stochastic differential \axiom{bsd} by
         ++ \axiom{introduce!}
```
◇

File defined by scraps 7, 8ab, 9abc, 10abc.

We begin by setting up the private tables and the various conversions. Note that the conversion from `Symbol` to `BSD` will *fail* if the relevant symbol has not yet been introduced (*AXIOM* determines whether or not this has been done by inspecting the private set `setBSD`).

Implementation is simplified since the conversion from `BSD` to `Symbol` can be carried out by code inherited from `Symbol` (the `ConvertibleTo(Symbol)` declaration enables this!):

"bsd.spad" 9c ≡

```
  Implementation ==> Symbol add

   Rep := Symbol

   setBSD := empty()$Set(Symbol)
   tableIto:Table(Symbol,%) := table()
   tableBSD:Table(%,Symbol) := table()

   convertIfCan(ds:Symbol):Union(%,"failed") ==
    not(member?(ds,setBSD)) => "failed"
    ds::%

   convert(ds:Symbol):% ==
    (du:=convertIfCan(ds))
```

```
     case "failed" =>
      print(hconcat(ds::Symbol::OF,
        message(" is not a stochastic differential")$OF))
      error "above causes failure in convert$BSD"
    du
◇
```

File defined by scraps 7, 8ab, 9abc, 10abc.

Introduction of a new basic stochastic differential is combined with a link to a semimartingale which yields the BSD as stochastic differential. Note that the semimartingale is not unique: it is determined up to its initial value (at some initial time which we do not specify in the current implementation). Note also that, as mentioned above, we do not have a type for semimartingale quantities. Consequently in our formal calculations the semimartingale property is left entirely implicit.

"bsd.spad" 10a ≡

```
   introduce!(X,dX) ==
    member?(dX,setBSD) => "failed"
    insert!(dX,setBSD)
    tableBSD(dX::%) := X
    tableIto(X) := dX::%
◇
```

File defined by scraps 7, 8ab, 9abc, 10abc.

After this it is appropriate to introduce a primitive Itô differential operating on symbols

"bsd.spad" 10b ≡

```
   d(X) ==
    search(X,tableIto) case "failed" => 0::INT
    tableIto(X)
◇
```

File defined by scraps 7, 8ab, 9abc, 10abc.

We will later need to inspect copies of the private records of the above transactions.

"bsd.spad" 10c ≡

```
   copyBSD() == [ds::% for ds in members(setBSD)]
   copyIto() == tableIto
   getSmgl(ds:%):Union(Symbol,"failed") == tableBSD(ds)
◇
```

File defined by scraps 7, 8ab, 9abc, 10abc.

Note that we do not here allow the private sets `setBSD`, `tableIto`, `tableBSD` to be reset: the reason being that it is safer simply either to `)clear completely` or even to exit *AXIOM* and start again! It is important that the user exercises considerable care in what they declare to be a semimartingale: consider the following example.

Example 2.1 : It is an mathematical error to invoke both of

- `introduce!(B,dB)`
- `introduce!(C,dC)`

where B represents a standard Brownian motion $t \mapsto B_t$ and C represents a time-displacement of this: $t \mapsto C_t = B_{t+1}$. It is not possible for both B and C to be semimartingales. For consider:

$$\mathbb{E}\left[B_{t+1}|\mathcal{F}_t\right] \quad = \quad \mathbb{E}\left[C_t|\mathcal{F}_t\right] \quad = \quad B_{t+1}$$

so the Doob-Meyer decomposition of B would have to fail!

Mistakes can be made in other ways as well. Considerable care must be exercised in declaring new semimartingales if proper meaning is to be carried by the formal calculations which follow. A safe rule is only to declare semimartingales once (using `introduce!`) at the start of the session, and then to ensure that the second-order structure of drifts and quadratic variations (as described below in Section 3) is defined consistently. However it is safe to declare further semimartingales so long as consistency is carefully checked.

We must test the above code. First we load the `BSD` library.

"bsd.input" 11a ≡

```
-- bsd.input
⟨revision and author information 2a⟩
-- Description:
-- Sequence of tests of basic properties of domain BSD.
⟨silence please 2c⟩
)library BSD
◇
```

File defined by scraps 11abc, 12abcd.

We now carry out the following basic steps.

1. Create a basic stochastic differential using `introduce!` and check printing properties and domain.

   "bsd.input" 11b ≡

   ```
   print copyBSD(); print copyIto()
   dX:=introduce!(X,dX) -- dX is a new stochastic differential
   print copyBSD(); print copyIto(); getSmgl(dX)
   print dX; print domainOf(dX);
   ◇
   ```

   File defined by scraps 11abc, 12abcd.

   Notice the useful idiom `dX:=introduce!(X,dX)`, which ensures that future reference to `dX` refers to an element of `BSD`.

2. Check the action of the primitive Itô differential on `BSD` and `Symbol`. An alternative is to package-call `d`.

   "bsd.input" 11c ≡

```
print d X;
print ((d$BSD) X); print domainOf((d$BSD) X);
◇
```

File defined by scraps 11abc, 12abcd.

3. Check that we can't create a new BSD two times over! (The second invocation of `introduce!(X,dX)` should yield `"failed"`.)

   "bsd.input" 12a ≡
   ```
   introduce!(t,dt) -- dt is a new stochastic differential
   print copyBSD(); print copyIto();
   introduce!(t,dt) -- fails (can't repeat an introduction!)
   print copyBSD(); print copyIto();
   ◇
   ```
   File defined by scraps 11abc, 12abcd.

4. Check that we can't simply convert `Symbol` or `String` to `BSD`, and check we can recapture a basic semimartingale using `getSmgl`:

   "bsd.input" 12b ≡
   ```
   'dZ::BSD          -- can fail
   dZ::BSD           -- fails
   string(dY)::BSD   -- fails
   print domainOf(dt)
   dt::BSD           -- succeeds!
   getSmgl(dt::BSD)
   ◇
   ```
   File defined by scraps 11abc, 12abcd.

   (the last two steps succeed because of the previous introduction of `dt`).

5. Check that we *can* convert from `BSD` to `Symbol`, and note that conversions to `String` must go by way of *explicit* conversion to `Symbol`:

   "bsd.input" 12c ≡
   ```
   dX::Symbol         -- succeeds
   string(dX)         -- fails
   string(dX::Symbol) -- succeeds
   ◇
   ```
   File defined by scraps 11abc, 12abcd.

6. The basic operations work for subscripted symbols:

   "bsd.input" 12d ≡
   ```
   [introduce!(A[i],dA[i]) for i in 1..2]
   print copyBSD(); print copyIto();
   [d A[i] for i in 1..2]
   ◇
   ```
   File defined by scraps 11abc, 12abcd.

   however some care needs to be exercised, since one cannot assign to a subscripted symbol. Hence the idiom `dA[3]:=introduce!(U,dU)` will not be accepted by *AXIOM*. Furthermore `A[i]` will not refer to a generic member of A_1, A_2, A_3, A_4, … . (These are intrinsic features of *AXIOM*.)

Chapter 3

Stochastic differentials

We are now able to use the above to implement a fundamental algebraic construction of stochastic calculus: the module (to be precise, algebra without unity) of *stochastic differentials*. We add a helper package SD2 so as to extend various operations to coerce BSD to SD(INT).

"sd.spad" 13 ≡

```
-- sd.spad
⟨revision and author information 2a⟩
)abbrev domain SD StochasticDifferential
--% StochasticDifferential
-- Compilation order:
-- BSD SD SD2 ITO ITO2 ITOINT OLDITO
--     **
++ Basic Operations: *, drift, alterQuadVar!, alterDrift!
++                      freeOf?, coefficient,
++                      copyQuadVar, copyDrift,
++                      uncorrelated?, statusIto()
++ Related Domains: BasicStochasticDifferential
++ Also See:
++ AMS Classifications:
++ Keywords: stochastic differential, semimartingale.
++ Examples:
++ References:
++ Ito (1975), Kendall (1991a,b; 1993a,b).
++ Description:
++ A basic implementation of
++ \axiom{StochasticDifferential(R)} using the
++ associated domain \axiomType{BasicStochasticDifferential}
++ in the underlying representation as sparse multivariate
++ polynomials. The domain is a module over \axiom{Expression(R)},
++ and is a ring without identity (AXIOM term is "Rng"). Note that
++ separate instances, for example using \axiom{R=Integer} and
++ \axiom{R=Float}, have different hidden structure (multiplication
++ and drift tables).
StochasticDifferential(R:Join(OrderedSet, IntegralDomain)):
 Category == Implementation where
  ER ==> Expression(R)
  PR ==> Polynomial(R)
```

```
  FR ==> Fraction(PR)
  BSD ==> BasicStochasticDifferential
  PI ==> PositiveInteger
  NNI ==> NonNegativeInteger
  OF  ==> OutputForm
  Category ==> Join(Rng, Module(ER)) with
```
◇

File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

This domain exports a number of operations:

- Retraction to BSD:

  "sd.spad" 14a ≡

  ```
      RetractableTo(BSD)
  ```
  ◇

  File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

  Note that retraction is fixed up automatically using the implementation.

- Add or alter a multiplication formula for two BSD:

  "sd.spad" 14b ≡

  ```
      alterQuadVar!:(BSD,BSD,%) -> Union(%,"failed")
       ++ alterQuadVar! adds multiplication formula for a
       ++ pair of stochastic differentials to a private table.
       ++ Failure occurs if
       ++  (a) either of first or second arguments is not basic
       ++  (b) third argument is not exactly of first degree
  ```
  ◇

  File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

- Add or alter a drift formula for a BSD argument:

  "sd.spad" 14c ≡

  ```
      alterDrift!:(BSD,%) -> Union(%,"failed")
       ++ alterDrift! adds drift formula for a
       ++ stochastic differential to a private table.
       ++ Failure occurs if
       ++  (a) first arguments is not basic
       ++  (b) second argument is not exactly of first degree
  ```
  ◇

  File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

- Obtain the relevant drift:

  "sd.spad" 14d ≡

  ```
      drift:% -> %
       ++ drift(dx) returns the drift of \axiom{dx}
  ```
  ◇

  File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

- Check whether a stochastic differential expression contains a given `BSD` as a module element (*not* in a coefficient *via* an Itô integral for example), also extract the coefficient of a given `BSD`, extract the list of component `BSD`, allow equations involving zeros, and display the resulting drift and multiplication tables:

  `"sd.spad"` 15a ≡

```
        freeOf?:(%,BSD) -> Boolean
         ++ freeOf?(sd,dX) checks whether \axiom{dX} occurs in
         ++ \axiom{sd} as a module element

        coefficient:(%,BSD) -> ER
         ++ coefficient(sd,dX) returns the coefficient of \axiom{dX}
         ++ in the stochastic differential \axiom{sd}

        listSD:(%) -> List BSD
         ++ listSD(dx) returns a list of all \axiom{BSD} involved
         ++ in the generation of \axiom{dx} as a module element

        equation:(%,R) -> Union(Equation %,"failed")
         ++ equation(dx,0) allows RHS of Equation % to be zero
        equation:(R,%) -> Union(Equation %,"failed")
         ++ equation(0,dx) allows LHS of Equation % to be zero

        copyDrift:() -> Table(%,%)
         ++ copyDrift returns private table of drifts
         ++ of basic stochastic differentials for inspection

        copyQuadVar:() -> Table(%,%)
         ++ copyQuadVar returns private multiplication table
         ++ of basic stochastic differentials for inspection
```
  ◇

  File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

- Multiplication by inverse:

  `"sd.spad"` 15b ≡

```
        "/"      : (%, ER) -> %
           ++ dx/y divides the stochastic differential dx
           ++ by the previsible function y.
```
  ◇

  File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

  Note that the above stretches the mathematical truth: the previsible functions constitute a ring not a field. However we are representing the previsible functions by `EXPR`, which *AXIOM* already deems to be a field, and furthermore we need to make sense of expressions such as `dX/a` . . . .

- Finally we need a function to display the current state of the private tables, and another to test whether stochastic differentials are uncorrelated:

"sd.spad" 16a ≡

```
statusIto:()  -> OF
 ++ statusIto() displays the current state of \axiom{setBSD},
 ++ \axiom{tableDrift}, and \axiom{tableQuadVar}. Question
 ++ marks are printed instead of undefined entries
uncorrelated?: (%,%) -> Boolean
 ++ uncorrelated?(dx,dy) checks whether its two arguments
 ++ have zero quadratic co-variation.
uncorrelated?: (List %,List %) -> Boolean
 ++ uncorrelated?(l1,l2) checks whether its two arguments
 ++ are lists of stochastic differentials of zero inter-list
 ++ quadratic co-variation.
uncorrelated?: (List List %) -> Boolean
 ++ uncorrelated?(ll) checks whether its argument is a list
 ++ of lists of stochastic differentials of zero inter-list
 ++ quadratic co-variation.
```

◇

File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

In this implementation (at version 1.6b and later) we ensure that drifts and stochastic differential multiplications are actually defined (rather than letting them default to yield 0, which is a strong statement related to assertions of independence, and which should be implied neither by ignorance nor by forgetfulness). First we base the implementation on sparse multivariate polynomials and add a way to divide by a predictable function:

"sd.spad" 16b ≡

```
Implementation ==> SparseMultivariatePolynomial(ER,BSD) add
 Rep:=SparseMultivariatePolynomial(ER,BSD)

 (v:% / s:ER):% == inv(s) * v
```

◇

File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

Now we define private multiplication and drift tables, and procedures to alter them.

"sd.spad" 16c ≡

```
tableQuadVar:Table(%,%)  := table()
tableDrift:Table(%,%) := table()

alterQuadVar!(da:BSD,db:BSD,dXdY:%):Union(%,"failed") ==
 -- next two lines for security only!
 1 < totalDegree(dXdY)                => "failed"
 0 ~= coefficient(dXdY,degree(1)$Rep) => "failed"
 not(0::% = (dXdY**2)::%) => "failed"
 setelt(tableQuadVar,((da::Rep)*(db::Rep))$Rep,dXdY)$Table(%,%)
 -- We have to take care here to avoid a bad
 -- recursion on \axiom{*:(%,%)->%}

alterDrift!(da:BSD,dx:%):Union(%,"failed") ==
 1 < totalDegree(dx)                => "failed"
```

```
 0 ~= coefficient(dx,degree(1)$Rep) => "failed"
 not(0::% = (dx**2)::%) => "failed"
 setelt(tableDrift,da::Rep,dx)$Table(%,%)
```
◇

File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

Now we define multiplication together with a couple of private functions.

"sd.spad" 17a ≡

```
multSDOrError(dm:%):% ==
 c := leadingCoefficient dm
 (dmm := search(dm/c,tableQuadVar))
   case "failed" =>
    print hconcat(message("ERROR IN ")$OF,(dm/c)::OF)
    error "Above product of sd's is not defined"
 c*dmm

(dx:% * dy:%) : % ==
 1 < totalDegree(dx) =>
  print hconcat(message("ERROR IN ")$OF,dx::OF)
  error "bad sd in lhs of sd product"
 1 < totalDegree(dy) =>
  print hconcat(message("ERROR IN ")$OF,dy::OF)
  error "bad sd in rhs of sd product"
 reduce("+",map(multSDOrError,monomials((dx*dy)$Rep)),0)
 -- We have to take care here to avoid a bad
 -- recursion on \axiom{*:(%,%)->%}
```
◇

File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

Next we deal with the `drift` function, which requires a similar private function.

"sd.spad" 17b ≡

```
driftSDOrError(dm:%):% ==
 c := leadingCoefficient dm
 (dmm := search(dm/c,tableDrift))
   case "failed" =>
    print hconcat(message("ERROR IN ")$OF,(dm/c)::OF)
    error "drift of sd is not defined"
 c*dmm

drift(dx:%):% ==
 reduce("+",map(driftSDOrError,monomials(dx)),0)
```
◇

File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

We add sundry utilities: a `freeOf?` function, a `coefficient` function, a function to obtain the list of component BSD, equations involving zeros, and two functions to display the private tables giving the current second-order structure (drifts and multiplications)

"sd.spad" 17c ≡

```
freeOf?(sd,dX) == (0 = coefficient(sd,dX,1))

coefficient(sd:%,dX:BSD):ER ==
 retract(coefficient(sd,dX,1))@ER

listSD(sd) ==
 [retract(dX)@BSD for dX in primitiveMonomials(sd)]

equation(dx:%,zero:R):Union(Equation %,"failed") ==
 not(0 = zero) => "failed"
 equation(dx,0::%)
equation(zero:R,dx:%):Union(Equation %,"failed") ==
 not(0 = zero) => "failed"
 equation(0::%,dx)

copyDrift() == tableDrift
copyQuadVar() == tableQuadVar
```
◇

File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

Finally we add the implementation of the function `statusIto()`, which displays the current state of the private tables, and `uncorrelated?`, which tests for uncorrelated stochastic differentials. Note that `statusIto()` uses a couple of convenient private functions.

"sd.spad" 18 ≡

```
xDrift(dx:BSD):OF ==
 (xdx := search(dx::Rep,tableDrift)) case "failed" => "?"::OF
 xdx::OF

xQV(dx:BSD,dy:BSD):OF ==
 (xdxdy := search((dx::% * dy::%)$Rep,tableQuadVar))
   case "failed" => "?"::Symbol::OF
 xdxdy::OF

statusIto():OF ==
 bsd  := copyBSD()$BSD
 bsdo := [dx::OF for dx in bsd]
 blank:= ""::Symbol::OF
 colon:= ":"::Symbol::OF
 bsdh := "BSD"::Symbol::OF
 dfth := "Drift"::Symbol::OF
 qvh  := "*"::Symbol::OF
 head := append([bsdh,colon],bsdo)
 drift:= append([dfth,colon],[xDrift dx for dx in bsd])
 space:= append([qvh ,blank],[blank for dx in bsd])
 qv   := [append([dy::OF,colon],[xQV(dx,dy) for dx in bsd])
                                            for dy in bsd]
 matrix(append([head,drift,space],qv))$OF
```
◇

File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

Here is the code for the test procedures `uncorrelated?`: note use of the private function `uncorrelated1?` to help break down a list of lists in a proper way.

"sd.spad" 19a ≡

```
uncorrelated?(dx:%,dy:%): Boolean == (0::% = dx*dy)

uncorrelated?(l1:List %,l2:List %): Boolean ==
 reduce("and", [
  reduce("and",[uncorrelated?(dx,dy) for dy in l2],true)
   for dx in l1 ],true)

uncorrelated1?(l1:List %,ll:List List %): Boolean ==
 reduce("and",[uncorrelated?(l1,l2) for l2 in ll],true)

uncorrelated?(ll:List List %): Boolean ==
 (0$Integer = # ll) => true
 (1 = # ll) => true
 uncorrelated1?(first ll,rest ll) and uncorrelated?(rest ll)
```

◇

File defined by scraps 13, 14abcd, 15ab, 16abc, 17abc, 18, 19a.

We now add a package SD2 to extend BSD to SD(INT) by default for various important operations:

"sd2.spad" 19b ≡

```
 --% sd2.spad
⟨revision and author information 2a⟩
)abbrev package SD2 StochasticDifferentialHelper
--% StochasticDifferentialHelper
-- Compilation order:
-- BSD SD SD2 ITO ITO2 ITOINT OLDITO
--        ***
++ Basic Operations: uncorrelated!
++ Related Domains: BasicStochasticDifferential, StochasticDifferential
++ Description:
++ A helper package for StochasticDifferential. It provides for
++ extension of \axiom{BSD} to \axiom{SD(INT)} for various important
++ operations.
StochasticDifferentialHelper(): Exports == Implementation where
 BSD ==> BasicStochasticDifferential
 SDI ==> StochasticDifferential(Integer)
 EXI ==> Expression(Integer)
 PI  ==> PositiveInteger
 Exports ==> with
```

◇

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

- The first pair of extensions deal with the cases when the third argument in alterQuadVar! is either BSD or 0:

  "sd2.spad" 19c ≡

  ```
  alterQuadVar!: (BSD,BSD,BSD) -> Union(SDI,"failed")
   ++ alterQuadVar!(dx:BSD,dy:BSD,dxdy:BSD) yields special
   ++ coercion of its third argument to \axiom{SD(INT)}
  alterQuadVar!: (BSD,BSD,Integer) -> Union(SDI,"failed")
  ```

```
   ++ alterQuadVar!(dx:BSD,dy:BSD,zero:Integer) yields special
   ++ coercion of its third argument to \axiom{SD(INT)},
   ++ but ONLY if this third argument equals \axiom{0}
```
◇

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

We add convenient procedures to declare BSD uncorrelated and to test for this:

"sd2.spad" 20a ≡

```
  uncorrelated!: (BSD,BSD) -> Boolean
   ++ uncorrelated!(dx,dy) declares its two arguments
   ++ to have zero quadratic co-variation.
  uncorrelated!: (List BSD,List BSD) -> Boolean
   ++ uncorrelated!(l1:,l2) declares its two arguments
   ++ to be lists of \axiom{BSD} of zero inter-list
   ++ quadratic co-variation.
  uncorrelated!: (List List BSD) -> Boolean
   ++ uncorrelated!(ll) declares its argument to be a list
   ++ of lists of \axiom{BSD} of zero inter-list
   ++ quadratic co-variation.
  uncorrelated?: (BSD,BSD) -> Boolean
   ++ uncorrelated?(dx,dy) checks whether its two
   ++ arguments have zero quadratic co-variation.
  uncorrelated?: (List BSD,List BSD) -> Boolean
   ++ uncorrelated?(l1,l2) checks whether its two arguments
   ++ are lists of basic stochastic differentials of zero
   ++ inter-list quadratic co-variation.
  uncorrelated?: (List List BSD) -> Boolean
   ++ uncorrelated?(ll) checks whether its argument is a list
   ++ of lists of basic stochastic differentials of zero
   ++ inter-list quadratic co-variation.
```
◇

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

- The second pair is very similar, but concern the second argument of alterDrift!:

"sd2.spad" 20b ≡

```
  alterDrift!:(BSD,BSD) -> Union(SDI,"failed")
   ++ alterDrift!(dx:BSD,Edx:BSD) yields special coercion of
   ++ its second argument to \axiom{SD(INT)}
  alterDrift!:(BSD,Integer) -> Union(SDI,"failed")
   ++ alterDrift!(dx:BSD,zero:Integer) yields special coercion
   ++ of its second argument to \axiom{SD(INT)}, but
   ++ ONLY if this second argument equals \axiom{0}
```
◇

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

- We need extensions of drift for those occasions when it is given a BSD or 0 argument:

"sd2.spad" 21a ≡

```
drift:(BSD) -> SDI
 ++ drift(dx:BSD) yields special coercion of its argument
 ++ so \axiom{SD(INT)}
drift:(Integer) -> Union(SDI,"failed")
 ++ drift(dx:BSD) yields special coercion of its argument
 ++ so \axiom{SD(INT)}
```
◇

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

- We need extensions of `*` *etc*, because we will want to carry out `Module` and `Rng` operations on `BSD` members from time to time.

"sd2.spad" 21b ≡

```
"*":(BSD,BSD) -> SDI
 ++ dx*dy extending multiplication to \axiom{BSD}
"*":(EXI,BSD) -> SDI
 ++ dx*dy extending multiplication to \axiom{BSD}
"*":(BSD,EXI) -> SDI
 ++ dx*dy extending multiplication to \axiom{BSD}

"/":(BSD,EXI) -> SDI
 ++ dx/y extending division to \axiom{BSD}

"^":(BSD,PI)  -> SDI
 ++ dx^n extending exponentiation (by positive integers
 ++ only!) to \axiom{BSD}

"+":(BSD,BSD) -> SDI
 ++ dx+dy extending addition to \axiom{BSD}
"-":(BSD,BSD) -> SDI
 ++ dx-dy extending subtraction to \axiom{BSD}
"-": BSD      -> SDI
 ++ -dx extending negation to \axiom{BSD}
```
◇

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

- Finally we need equations using `BSD` which involve zeros which must be coerced from `Integer` to `SD(INT)`:

"sd2.spad" 21c ≡

```
equation:(BSD,Integer) -> Union(Equation SDI,"failed")
 ++ equation(dx,0) to coerce to \axiom{Equation SDI}
equation:(Integer,BSD) -> Union(Equation SDI,"failed")
 ++ equation(0,dx) to coerce to \axiom{Equation SDI}
```
◇

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

The implementation is *very* mundane!

"sd2.spad" 22a ≡

```
 Implementation ==> add
  alterQuadVar!(dx:BSD,dy:BSD,dxdy:BSD):Union(SDI,"failed") ==
   alterQuadVar!(dx,dy,dxdy::SDI)$SDI
  alterQuadVar!(dx:BSD,dy:BSD,zero:Integer):Union(SDI,"failed") ==
   not(0 = zero) => "failed"
   alterQuadVar!(dx,dy,0::SDI)$SDI
```

◇

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

Here is the code for the constructor procedures `uncorrelated!`:

"sd2.spad" 22b ≡

```
  uncorrelated!(dx:BSD,dy:BSD):Boolean ==
   alterQuadVar!(dx,dy,0::SDI)$SDI case SDI

  uncorrelated!(l1:List BSD,l2:List BSD): Boolean ==
   for dx in l1 repeat
    for dy in l2 repeat
     uncorrelated!(dx,dy)
   true

  uncorrelated!(ll:List List BSD): Boolean ==
   (0$Integer = # ll) => false
   (1 = # ll) => true
   for l2 in rest ll repeat
     uncorrelated!(first ll,l2)
   uncorrelated!(rest ll)
   true
```

◇

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

Here is the code to ensure the test procedures `uncorrelated?` work with basic stochastic differentials: some repetition of the `sd.spad` code is required because of problems coercing lists of basic stochastic differentials to lists of stochastic differentials, *etc.*

"sd2.spad" 22c ≡

```
  uncorrelated?(dx:BSD,dy:BSD): Boolean ==
   uncorrelated?(dx::SDI,dy::SDI)

  uncorrelated?(l1:List BSD,l2:List BSD): Boolean ==
    reduce("and", [
     reduce("and", [uncorrelated?(dx,dy) for dy in l2])
      for dx in l1 ])

  uncorrelated1?(l1:List BSD,ll:List List BSD): Boolean ==
   reduce("and",[uncorrelated?(l1,l2) for l2 in ll],true)

  uncorrelated?(ll:List List BSD): Boolean ==
    (0$Integer = # ll) => true
    (1 = # ll) => true
```

```
          uncorrelated1?(first ll,rest ll) and uncorrelated?(rest ll)
◇
```

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

"sd2.spad" 23a ≡

```
        alterDrift!(dx:BSD,Edx:BSD):Union(SDI,"failed") ==
         alterDrift!(dx,Edx::SDI)$SDI
        alterDrift!(dx:BSD,zero:Integer):Union(SDI,"failed") ==
         not(0 = zero) => "failed"
         alterDrift!(dx,0::SDI)$SDI

        drift(dx:BSD):SDI == drift(dx::SDI)
        drift(zero:Integer):Union(SDI,"failed") ==
         not(0 = zero) => "failed"
         drift(0::SDI)

        (dx:BSD * dy:BSD):SDI == (dx::SDI) * (dy::SDI)
        (x:EXI  * dy:BSD):SDI ==         x * (dy::SDI)
        (dx:BSD * y:EXI):SDI  == (dx::SDI) * y

        (dx:BSD / y:EXI):SDI  == (dx::SDI) / y

        (dx:BSD ^ n:PI):SDI  == (dx::SDI) ** n

        (dx:BSD + dy:BSD):SDI == (dx::SDI) + (dy::SDI)
        (dx:BSD - dy:BSD):SDI == (dx::SDI) - (dy::SDI)

        (- dx:BSD):SDI        == - (dx::SDI)

        equation(dx:BSD,zero:Integer):Union(Equation SDI,"failed") ==
         not(0 = zero) => "failed"
         equation(dx::SDI,0::SDI)
        equation(zero:Integer,dx:BSD):Union(Equation SDI,"failed") ==
          not(0 = zero) => "failed"
          equation(0::SDI,dx::SDI)
◇
```

File defined by scraps 19bc, 20ab, 21abc, 22abc, 23a.

Again we must test the domain SD(INT) and associated package SD2. We set up basic stochastic differentials dt, dX, dY.

"sd.input" 23b ≡

```
-- sd.input
⟨revision and author information 2a⟩
-- Description:
-- Sequence of tests of domain SD and auxiliary package SD2.
⟨silence please 2c⟩
)library BSD SD SD2

dt := introduce!(t,dt)
dX := introduce!(X,dX)
dY := introduce!(Y,dY)
copyBSD()
copyIto()
◇
```

File defined by scraps 23b, 24abc, 25ab, 26abc.

We set the quadratic variations and drifts individually, to implement `dt` as the time differential and `dX`, `dY` as independent Brownian differentials.

"sd.input" 24a ≡
```
alterQuadVar!(dt,dt,0)
alterQuadVar!(dX,dt,0)
alterQuadVar!(dt,dY,0)

alterQuadVar!(dY,dX,dt)
```
◇

File defined by scraps 23b, 24abc, 25ab, 26abc.

Later we package up these tasks as procedures, but for now we work through it all in detail.

"sd.input" 24b ≡
```

alterQuadVar!(dX,dX,dt)
alterQuadVar!(dY,dY,dX)    -- fails (dX not classical)
alterQuadVar!(dY,dY,2*dt) -- succeeds

copyQuadVar()              -- display of multiplication table
[[da*db for db in copyBSD()] for da in copyBSD()] -- compare!
```
◇

File defined by scraps 23b, 24abc, 25ab, 26abc.

Here are brief tests of `uncorrelated!`:

"sd.input" 24c ≡
```
alterQuadVar!(dt,dt,0)
uncorrelated!([dt],[dX,dY])
uncorrelated?([dt],[dX,dY])
uncorrelated?([dt,dX],[dX,dY])

uncorrelated!(dX,dY); statusIto()
uncorrelated?(dX,dY)
uncorrelated?(dY,dY)

uncorrelated!([[dt],[dX],[dY]]); statusIto()
uncorrelated?([[dt],[dX],[dY]])
uncorrelated?([[dt],[dX],[dX]])
```
◇

File defined by scraps 23b, 24abc, 25ab, 26abc.

Caution 1 *Notice it is necessary to ensure that the specified quadratic variations are coherent. This is exactly to say, the process*

$$Q(a) \quad = \quad \sum_i \sum_j a_i a_j \frac{\mathtt{d}[X_i, X_j]}{\mathtt{d}t} \tag{3.1}$$

(summing over the basic semimartingales X_i) must be almost surely a non-negative definite quadratic form in the vector argument a. There is no clean way to enforce this analytic fact within the AXIOM computer algebra package

(this is an analytic rather than algebraic problem!); it is therefore the user's responsibility to make sure that the specified quadratic variations do not violate the above.

Drifts are similar:

"sd.input" 25a ≡
```
alterDrift!(dt,dt)
alterDrift!(dX,0)
alterDrift!(dY,0)
alterDrift!(dY,dX) -- fails (dX not classical)

copyDrift()       -- display of drift list
[drift(dx) for dx in copyBSD()] -- compare!
```
◇

File defined by scraps 23b, 24abc, 25ab, 26abc.

Caution 2 *Again it is necessary to ensure a consistency condition. This time we must ensure that a semimartingale with zero quadratic variation (yielding a "classical" differential) has differential left unchanged by* `drift`*:*

$$\mathrm{d}[X,X] \quad = \quad 0 \qquad \textit{implies} \qquad \texttt{drift}\,\mathrm{d}X \quad = \quad 0\,. \tag{3.2}$$

Again, there is no economical way to perform this automatically within AXIOM (note that the relationship must hold for all *semimartingales we can generate, not just the basic semimartingales!) and so it is left as the user's responsibility to ensure that this consistency condition holds.*

As long as mild regularity prevails, these conditions (3.1), (3.2) suffice to ensure that the resulting semimartingales are well-defined.

Note that a difficulty remains if we want to work with both `SD INT` and - for example - `SD FLOAT`. For then the two domains would use *separate* private drift and multiplication tables. Coercions from `BSD` would become ambiguous, except that we have disambiguated them by moving relevant implicit coercions (overloaded procedures) into the helper package `SD2`!

We now check various basic combinations of stochastic differentials.

"sd.input" 25b ≡
```
print (2*dX);
print (dX*dY); print (dX^2); print (dX**2);
print (dY*dX); print (dY^2); print (dY**2);

print (dX^3); print ((dX^2)*dt);

print (dX + dY); print (dX - dY); print (- dY);

dZ := sin(X)*dX

print (dZ/3); print (dZ/cos(X)); print (dZ/sin(X));
print (dX + dZ); print (-dZ)
```
◇

File defined by scraps 23b, 24abc, 25ab, 26abc.

Now check some pathologies, which ought to fail!

"sd.input" 26a ≡

```
sin(dZ) -- fails
1/dZ    -- fails
dZ^0    -- fails
dZ + 1  -- fails
◇
```

File defined by scraps 23b, 24abc, 25ab, 26abc.

We now display a multiplication table. Note that we can't construct matrices with SD(INT) entries, because there is no identity in SD(INT) (nor in SD(R) for any other suitable ring R).

"sd.input" 26b ≡

```
nbsd := #copyBSD();
ItoMultArray:ARRAY2(SD INT) :=new(nbsd,nbsd,0$SD(INT));
sdmult :=
 [[(dx*dy) for dx in copyBSD()] for dy in copyBSD()];
for i in 1..(nrows(ItoMultArray)) repeat
 for j in 1..(nrows(ItoMultArray)) repeat
  ItoMultArray(i,j) := sdmult(i)(j)
ItoMultArray

statusIto()
◇
```

File defined by scraps 23b, 24abc, 25ab, 26abc.

Finally some checks on basic Itô calculus, and testing `freeOf?`, `coefficient`, `equation`:

"sd.input" 26c ≡

```
print drift dX; print drift dY; print drift dt;

dw := dZ + dY + dt
print (dw^2); print drift dw; print drift (dw^2);

freeOf?(dw,dt)
freeOf?(dw-dt,dt)

coefficient(dw,dX)
[coefficient(dw,dx) for dx in listSD(dw)]
dw - reduce(+,[coefficient(dw,dx)*dx for dx in listSD(dw)])

print (dt = dw); print (0 = dw); print (dw = 0);
◇
```

File defined by scraps 23b, 24abc, 25ab, 26abc.

Chapter 4

Implementation of the Itô differential

The fundamental operation `d` of the Itô differential is currently implemented as a mapping from an ordinary `Expression(R)` to a `SD(R)` object (for some parameter domain `R` - typically `INT` or `FLOAT`). Our implementation works with the structure of a semimartingale expression viewed as an `EXPR INT`. The major piece of work is to implement the second-order Itô formula.

In the literature there are many proofs of the Itô formula. In [7] it is proved in a way which emphasizes its geometric and second-order aspects.

Here is the resulting package `Ito`.

"ito.spad" 27 ≡

```
-- ito.spad
⟨revision and author information 2a⟩
)abbrev package ITO Ito
--% Ito
-- Compilation order:
-- BSD SD SD2 ITO ITO2 ITOINT OLDITO
--             ***
++ Basic Operations: d, smooth!
++ Related Domains: BasicStochasticDifferential,
++                  StochasticDifferential
++ Also See:
++ AMS Classifications:
++ Keywords: stochastic differential, semimartingale.
++ Examples:
++ References:
++ Many! But also see Kendall (1992) for a geometric treatment based
++ on second-order approximation.
++ Description:
++ Implements the stochastic differential.
++
Ito(R:Join(OrderedSet, IntegralDomain)):
 Exports == Implementation where
  BOP  ==> BasicOperator
  LBO  ==> List(BOP)
  FR   ==> Fraction(Polynomial(R))
```

```
EXR  ==> Expression(R)
KEXR ==> Kernel(EXR)
SMP  ==> SparseMultivariatePolynomial
SMPK ==> SMP(R,KEXR)
OF   ==> OutputForm
BSD  ==> BasicStochasticDifferential
SDR  ==> StochasticDifferential(R)
RC1  ==> Record(e:EXR,i:SDR)
RC2  ==> Record(s:Symbol,e:EXR,i:SDR)
```
◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

There are just two exports: the Itô differential `d` and a procedure `smooth!` to declare operators to be sufficiently smooth. Note that the (old-style) *AXIOM* compiler returns warnings for `d`, `ItoDExpt`, `ItoDKernel2`, ... , which I have not yet been able to circumvent.

"ito.spad" 28a ≡

```
Exports ==> with
 d:EXR -> SDR
  ++ d(X) yields the stochastic differential of \axiom{X}
 d:FR -> SDR
  ++ d(X) yields the stochastic differential of \axiom{X}
 smooth!:BOP -> Union(BOP,"failed")
  ++ smooth!(op) attaches property "ito" to operator \axiom{op}
```
◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

Examining the *AXIOM* definition of an `Expression`, we discover a recursion. A `Expression` is given by a fraction for which numerator and denominator are both in the form of a `Polynomial` of `Kernels`; while a `Kernel` is either a constant, a variable, or a function of several `Expressions`. We therefore deal with the implementation using a family of inter-related procedures:

"ito.spad" 28b ≡

```
Implementation ==> add
 ItoDKernel:KEXR        -> Union(SDR,"failed")
 ItoDExpt:SMPK          -> SDR
 ItoDRoot:KEXR          -> SDR
 ItoDMono:SMPK          -> SDR
 ItoDPoly:SMPK          -> SDR
 ItoDKernel2:KEXR       -> SDR
 ItoDKernelk:KEXR       -> SDR
```
◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

We start by considering the case of the Ito differential of a `Kernel`. We have to take account of the differentiability of the function at the top level of the kernel: to apply the Itô formula to a non-trivial kernel we need to know that the relevant operators are smooth (in fact, C^2). For simplicity in this first try, we deal only with the unary case. We store the names of common (locally) smooth operators in `setSmooth`: I would prefer to set the `"ito"` property from within the package, but can't figure out how to do this. For less common operators, use `smooth!`.

"ito.spad" 29a ≡

```
smooth!(op:BOP):Union(BOP,"failed")==
 assert(op,"ito")

setSmooth:={
 'exp,'log,'sin,'cos,'tan,
  'cot,'sec,'csc,'asin,'acos,'atan,
   'acot,'asec,'acsc,'sinh,'cosh,'tanh,
    'coth,'sech,'csch,'asinh,'acosh,'atanh,
     'acoth,'asech,'acsch,'Gamma
      }::Set Symbol
```

◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

We now turn to actual calculation of the Itô differential. First we deal with a full `Expression X`. We begin by testing for whether the `height` vanishes, in which case `X` is a constant and the differential vanishes. Otherwise we divide `X` into numerator and denominator, each of which are `Polynomial`. (Note that we add a further procedure to convert fractions of polynomials into expressions!)

"ito.spad" 29b ≡

```
d(X:EXR):SDR ==
 0 = height X => 0$SDR
 f  := numerator X
 g  := denominator X
 df := ItoDPoly(numer f)
 dg := ItoDPoly(numer g)
 (g*df - f*dg)/g**2 - df*dg/g**2 + f*dg**2/g**3

d(X:FR):SDR == d(X::EXR)
```

◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

We now have to process a `Polynomial` of `Kernels`: this is a simple matter of summing an application of a further procedure `ItoDMono` to each constituent monomial.

"ito.spad" 29c ≡

```
ItoDPoly(X:SMPK):SDR ==
 reduce("+",[ItoDMono m for m in monomials(X)],0$SDR)
```

◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

The next step is to deal with monomials. General monomials have to be broken up into a product of powers of kernels. A monomial which is a single power has to be treated separately. We deal with the product term-by-term. In principle we could use binary divide-and-conquer, but this would involve tricky programming. We could also include code to detect when the second-order terms would automatically vanish, but this too would increase code-complexity. So we now work term-by-term

"ito.spad" 30a ≡

```
ItoDMono(X:SMPK):SDR ==
 (lX := isTimes(X)) case "failed" => ItoDExpt(X)
 dlX := [ItoDExpt(pX)/(pX::EXR) for pX in lX]
 (X::EXR)*(reduce("+",dlX,0$SDR)
  + reduce("+",
   [reduce("+",[dlX(i)*dlX(j)
    for j in (i+1)..#(lX)],0$SDR)
     for i in 1..#(lX)],0$SDR))
```

◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

The simplest kind of monomial is a power of a `Kernel`. The next piece of code deals with a monomial which is a simple power of a kernel:

"ito.spad" 30b ≡

```
ItoDExpt(pX:SMPK):SDR ==
 (rpX := isExpt pX) case "failed" => 0$SDR
 rpX:=rpX::Record(var:KEXR,exponent:NonNegativeInteger)
 n := (rpX.exponent)::Integer
 x := rpX.var
 dx := ItoDKernel(x::KEXR)::SDR
 n*(x::EXR)**(n-1)*dx
  + (n*(n-1)::EXR/2::EXR)*(x::EXR)**(n-2)*dx**2
```

◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

Now we consider kernels. The easiest case is when the kernel is actually a symbol. In that case we can use the `BSD` function `d`. Otherwise we have to apply the Itô formula to the kernel. For simplicity in this first try, we deal only with the cases of one or two arguments. Note that exceptions have to be made for `itoIntegral` and also for the case when the kernel operator is a power.

"ito.spad" 30c ≡

```
ItoDKernel(X:KEXR):Union(SDR,"failed") ==
 (s := symbolIfCan X) case Symbol =>
  (ds:=(d s)$BSD) case Integer => 0::SDR
  ds::BSD::SDR
 f  := operator X
 ("itoIntegral"::Symbol) = name f =>
  (argument X)(1)*d((argument X)(2))
 ("%power"::Symbol) = name f => ItoDKernel2(X)
 ("nthRoot"::Symbol) = name f => ItoDRoot(X)
 not (has?(f,"ito") or member?(name(f),setSmooth))=>
  print(hconcat((name operator X)::OF,
   message(" not known to be smooth enough for Ito formula")$OF))
  "failed"
 (#(argument X) = 2) => ItoDKernel2(X)
 (#(argument X) > 2) => ItoDKernelk(X)
 -- print message("ItoDKernel: at most 2 arguments for now")$OF
 -- "failed"
 arg := first argument X
```

```
sd := d arg
x  := new()$Symbol
f0 := kernel(f,[x::EXR],1)::EXR
f1 := differentiate(f0,x)
f2 := differentiate(f1,x)
eval(f1,kernel(x),arg)$EXR*sd
 + (1::EXR/2::EXR)*eval(f2,kernel(x),arg)$EXR*sd**2
```

◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

It is convenient to deal separately with the potentially involved case when we wish to find the Itô differential of a power such as B^C or the Itô differential of an n^{th} root such as $\sqrt{B}$, $\sqrt{C}B$, or B^C.

"ito.spad" 31a ≡

```
ItoDKernel2(X:KEXR):SDR ==
 f := operator X
 a := (argument X)(1)
 b := (argument X)(2)
 da:= d a
 db:= d b
 x := new()$Symbol
 y := new()$Symbol
 f0 := kernel(f,[x::EXR,y::EXR],2)::EXR
 fx := differentiate(f0,x)
 fy := differentiate(f0,y)
 fxx:= differentiate(fx,x)
 fxy:= differentiate(fx,y)
 fyy:= differentiate(fy,y)
 eval(fx,[kernel(x),kernel(y)],[a,b])$EXR*da
  + eval(fy,[kernel(x),kernel(y)],[a,b])$EXR*db
   + (1::EXR/2::EXR)*(
    eval(fxx,[kernel(x),kernel(y)],[a,b])$EXR*da**2
     + 2*eval(fxy,[kernel(x),kernel(y)],[a,b])$EXR*da*db
      + eval(fyy,[kernel(x),kernel(y)],[a,b])$EXR*db**2)
```

◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

The case of a root has to be treated carefully since we have made no assumptions about the underlying coefficient domain R.

"ito.spad" 31b ≡

```
ItoDRoot(X:KEXR):SDR ==
 x := (argument X)(1)
 a := 1/((argument X)(2))
 dx := d x
 a*(X::EXR/x)*dx
  + (a*(a-1::EXR)/2::EXR)*(X::EXR/x**2)*dx**2
```

◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

Smooth functions of several arguments require their own special treatment. Here we omit one obvious economy in the interests of clarity of code: we do not exploit the symmetry of mixed derivatives.

"ito.spad" 32a ≡

```
ItoDKernelk(X:KEXR):SDR ==
 fn := operator X
 al := argument X
 vl := [[new()$Symbol,a,d(a)]@RC2 for a in al]
 kl := [kernel(v.s) for v in vl]
 f0 := kernel(fn,[(v.s)::EXR for v in vl],#(vl))::EXR
 f1 := [[differentiate(f0,v.s),v.i]@RC1 for v in vl]
 f2 := [[eval((differentiate(h.e,v.s)),kl,al)*(v.i)*(h.i)
         for v in vl] for h in f1]
 reduce("+",[eval((h.e),kl,al)*h.i for h in f1],0$SDR)
  +(1::EXR/2::EXR)*reduce("+",[reduce("+",hlist,0$SDR)
     for hlist in f2],0$SDR)
```

◇

File defined by scraps 27, 28ab, 29abc, 30abc, 31ab, 32a.

We now produce code for testing the above procedures. Notice we anticipate the next chapter and the appendix by calling on ITO2 and OLDITO. The test file starts as follows:

"ito.input" 32b ≡

```
-- ito.input
⟨revision and author information 2a⟩
-- Description:
-- Test of the ITO package.
⟨silence please 2c⟩
)library BSD SD SD2 ITO ITO2 ITOINT OLDITO
```

◇

File defined by scraps 32bc, 33abc.

For test purposes we need a triple of semimartingales: we choose time t and two correlated local martingales B and C. The following code introduces these and defines their second-order structure. Notice that we have fixed B, C to be entirely general!

"ito.input" 32c ≡

```
-- semimartingales for testing purposes
dt:=introduce!(t,dt);  classicalSD!(dt);
dB:=introduce!(B,dB);  brownianSD!(dB,a11*dt);
dC:=introduce!(C,dC);  brownianSD!(dC,a22*dt);
alterDrift!(dB,a1*dt); alterDrift!(dC,a2*dt);
alterQuadVar!(dB,dC,a12*dt);
```

◇

File defined by scraps 32bc, 33abc.

We now work through some test inputs. We begin with basic semimartingales, linear, polynomial, and fractional polynomial combinations:

"ito.input" 33a ≡

```
[d(s)-OldD(s) for s in
 [1, h, B, a*B+b, a*B^2+b^B+c, (a*B^2+b*B+c)/(a1*B^2+b1^B+c1)]]
[d(s)-OldD(s) for s in
 [B^4, B*C, a*B*C, 4*a*B*C, B^5*C^6, a*B^5*C^6, 5*a*B^5*C^6]]
x:=(h + 4*a*B*C + B^5*C^6); d(x)-OldD(x)
◇
```

File defined by scraps 32bc, 33abc.

The next step is to consider kernels, including general powers, square roots, and surds. Note that *AXIOM* is case-sensitive, and so in general $c \neq C$. Indeed, because we have note `introduced` c it is treated below as a constant parameter.

"ito.input" 33b ≡

```
[d(s)-OldD(s) for
 s in [sin(B), sin(4*h*B^3*C), sin(B)^2, 4*h*sin(B)^2*tan(C)]]
x:=(sin(sin(B))); d(x)-OldD(x)
[d(s)-OldD(s) for
 s in [B^C, b^C, B^c, t^t, B^t, t^C, sqrt{C}, C^{1/B}]]
◇
```

File defined by scraps 32bc, 33abc.

Now consider some exotic combinations:

"ito.input" 33c ≡

```
[d(s)-OldD(s) for
 s in [sin(B)^2/(1+sin(B)),sin(B)^B/(1+log(B)),sin(B)^(cos(C)+1)]]
◇
```

File defined by scraps 32bc, 33abc.

In a further, more realistic, test we introduce two independent Brownian motions `X` and `Y`, this time without using `ITO2`:

"bessel.input" 33d ≡

```
-- bessel.input
⟨revision and author information 2a⟩
-- Description:
-- Using the ITO package for the Bessel process
⟨silence please 2c⟩
)library BSD SD SD2 ITO

dt := introduce!(t,dt);  alterQuadVar!(dt,dt,0);
alterDrift!(dt,dt);

dX := introduce!(X,dX);  alterQuadVar!(dX,dX,dt);
alterDrift!(dX,0);
alterQuadVar!(dt,dX,0);

dY := introduce!(Y,dY);  alterQuadVar!(dY,dY,dt);
alterDrift!(dY,0);
alterQuadVar!(dt,dY,0);  alterQuadVar!(dX,dY,0);
◇
```

File defined by scraps 33d, 34ab, 35ab.

This allows us to construct a Bessel(2) process

$$R = \sqrt{(X^2 + Y^2)}$$

or, in *AXIOM*,

"bessel.input" 34a ≡
```
R:=(X^2+Y^2)^(1/2)
dR := d R

print (dR^2); print drift dR;
print ((drift dR) - (d t)/(2*R));
◇
```
File defined by scraps 33d, 34ab, 35ab.

with output from the last line as follows:
[Input: `R:=(X**2+Y**2)**(1/2)`]

$$\sqrt{Y^2 + X^2}$$

[Input: `dR := d R`]

$$\frac{1}{2\,\sqrt{Y^2 + X^2}}\ dt + \frac{Y}{\sqrt{Y^2 + X^2}}\ dY + \frac{X}{\sqrt{Y^2 + X^2}}\ dX$$

[Input: `dR**2`]

$$dt$$

[Input: `drift dR`]

$$\frac{1}{2\,\sqrt{Y^2 + X^2}}\ dt$$

[Input: `(drift dR) - (d t)/(2*R)`]

$$0$$

Finally consider a general approach to the Itô formula:

"bessel.input" 34b ≡
```

f := operator 'f;   smooth! f;

d f(X); print drift d f(X); print ((d f(X))^2);
◇
```
File defined by scraps 33d, 34ab, 35ab.

with output from the last line
[Input: `d f(X)`]

$$\frac{f''(X)}{2}\ dt + f'(X)\ dX$$

[Input: `drift d f(X)`]

$$\frac{f''(X)}{2}\ dt$$

[Input: `(d f(X))**2`]

$$f'(X)^2\ dt$$

or in two dimensions

"bessel.input" 35a ≡

```
print d f(X,Y); print drift d f(X,Y); print ((d f(X,Y))^2);
◇
```

File defined by scraps 33d, 34ab, 35ab.

yielding output

[Input: `d f(X,Y)`]

$$\frac{f_{,2,2}(X,\ Y) + f_{,1,1}(X,\ Y)}{2}\ dt + f_{,2}(X,\ Y)\ dY + f_{,1}(X,\ Y)\ dX$$

[Input: `drift d f(X,Y)`]

$$\frac{f_{,2,2}(X,\ Y) + f_{,1,1}(X,\ Y)}{2}\ dt$$

[Input: `(d f(X,Y))**2`]

$$\left(f_{,2}(X,\ Y)^2 + f_{,1}(X,\ Y)^2\right)\ dt$$

and finally in 3 dimensions

"bessel.input" 35b ≡

```
dZ := introduce!(Z,dZ);2  alterQuadVar!(dZ,dZ,dt);
alterDrift!(dZ,0);
alterQuadVar!(dt,dZ,0);
alterQuadVar!(dX,dZ,0);  alterQuadVar!(dY,dZ,0);

print d f(X,Y,Z); print drift d f(X,Y,Z); print ((d f(X,Y,Z))^2);
◇
```

File defined by scraps 33d, 34ab, 35ab.

[Input: `d f(X,Y,Z)`]

$$\begin{aligned} & \frac{f_{,3,3}(X,\ Y,\ Z) + f_{,2,2}(X,\ Y,\ Z) + f_{,1,1}(X,\ Y,\ Z)}{2}\ dt \\ + \quad & f_{,3}(X,\ Y,\ Z)\ dZ + f_{,2}(X,\ Y,\ Z)\ dY + f_{,1}(X,\ Y,\ Z)\ dX \end{aligned}$$

[Input: `drift d f(X,Y,Z)`]

$$\frac{f_{,3,3}(X,\ Y,\ Z) + f_{,2,2}(X,\ Y,\ Z) + f_{,1,1}(X,\ Y,\ Z)}{2}\ dt$$

[Input: `((d f(X,Y,Z))**2)`]

$$\left(f_{,3}(X,\ Y,\ Z)^2 + f_{,2}(X,\ Y,\ Z)^2 + f_{,1}(X,\ Y,\ Z)^2\right)\ dt$$

Chapter 5

Further procedures for the Itô formula package

Using the above procedures as basic building blocks, we can construct further procedures to refine the mathematical interface. We package these in the associated package ITO2.

First we need a way to install and test for classical differentials. Note that the test for a classical differential assumes consistency! We therefore do not test for zero quadratic co-variation with all other semimartingale differentials.

We intersperse the package code with elementary tests. To start with, here are declarations for ITO2 and the standard header for the test file.

"ito2.spad" 37 ≡

```
-- ito2.spad
⟨revision and author information 2a⟩
)abbrev package ITO2 Ito2
--% Ito2
-- Compilation order:
-- BSD SD SD2 ITO ITO2 ITOINT OLDITO
--                ****
++ Basic Operations: classicalSD!, classicalSD?,
++                   brownianSD!,brownianSD?,
++                   itoSDE!, itoSDE?
++ Related Domains: BasicStochasticDifferential,
++                  StochasticDifferential
++ Also See: Ito package (eventually to include this!)
++ AMS Classifications:
++ Keywords: stochastic differential, semimartingale.
++ Examples:
++ References:
++ Many! But also see Kendall (1992) for a geometric treatment based
++ on second-order approximation.
++ Description:
++ Implements the stochastic differential.
++
Ito2(R:Join(OrderedSet, IntegralDomain)): Exports == Implementation where
 BSD ==> BasicStochasticDifferential
 SDR ==> StochasticDifferential(R)
```

```
 Exports ==> with
```
◇

File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

1. Procedures to set up and test for classical stochastic differentials:

   "ito2.spad" 38a ≡

   ```
   classicalSD!:BSD -> SDR
    ++ classicalSD!(dt) sets up the \axiom{BSD} argument to
    ++ be a classical stochastic differential: zero quadratic
    ++ variations and co-variations (with currently existing
    ++ \axiom{BSD}) and invariant under \axiom{drift}
   classicalSD?:SDR -> Boolean
    ++ classicalSD?(dt) tests whether its \axiom{SDR} argument
    ++ is a classical stochastic differential. It DOESN'T test
    ++ co-variations (perhaps it should?!)
   classicalSD?:BSD -> Boolean
    ++ classicalSD?(dt) extended to \axiom{BSD} arguments
   ```
   ◇

   File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

2. Procedures to set up and test for (univariate) Brownian motions in random time-scales:

   "ito2.spad" 38b ≡

   ```
   brownianSD!:(BSD,SDR) -> Union(SDR,"failed")
    ++ brownianSD!(dB,dt) sets up \axiom{dB} as a Brownian
    ++ differential in the \axiom{dt} timescale. A test is
    ++ made as to whether \axiom{dt} is classical. The
    ++ relationship with NON_CLASSICAL differentials is left
    ++ open.
   brownianSD!:(BSD,BSD) -> Union(SDR,"failed")
    ++ brownianSD!(dB,dt) extended to \axiom{BSD} arguments
   brownianSD?:(SDR,SDR) -> Boolean
    ++ brownianSD?(dB,dt) tests whether \axiom{dB} is indeed a
    ++ Brownian motion in the \axiom{dt} time-scale.
   brownianSD?:(BSD,BSD) -> Boolean
    ++ brownianSD?(dB,dt) extended to \axiom{BSD} arguments
   ```
   ◇

   File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

3. A procedure to test whether a stochastic differential actually vanishes

   "ito2.spad" 38c ≡

   ```
   zeroSD?:(SDR) -> Boolean
    ++ zeroSD?(dX) tests whether \axiom{dX} is zero according
    ++ to its second-order structure
   ```
   ◇

   File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

4. Procedures to set up a semimartingale as the solution to a strong *sde*, and to test that it is such a solution:

"ito2.spad" 39a ≡

```
        itoSDE!:(Equation SDR) -> Union(SDR,"failed")
         ++ itoSDE!(sde) checks whether \axiom{sde} is of suitable
         ++ form (\axiom{lhs} is an unset \axiom{BSD} while \axiom{rhs}
         ++ is free of \axiom{lhs}) and then sets \axiom{lhs} accordingly.
        itoSDE!:(Equation BSD) -> Union(SDR,"failed")
         ++ itoSDE!(sde) extended to \axiom{BSD} arguments
        itoSDE?:(Equation SDR) -> Boolean
         ++ itoSDE?(sde) checks whether \axiom{sde} holds by the simple
         ++ resort of testing whether or not \axiom{(lhs sde)-(rhs sde)}
         ++ is zero in the SD module
        itoSDE?:(Equation BSD) -> Boolean
         ++ itoSDE?(sde) extended to \axiom{Equation BSD} arguments
```
◇

File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

5. A procedure to set up and test for a multivariate Brownian motion in a random time scale:

"ito2.spad" 39b ≡

```
        brownianSD!:(List BSD,SDR) -> Union(List BSD,"failed")
         ++ brownianSD!([dB, ...],dt) sets up \axiom{[dB,...]} as
         ++ a vector-valued Brownian differential in the \axiom{dt}
         ++ timescale, using the univariate version and
         ++ \axiom{uncorrelated}.
        brownianSD!:(List BSD,BSD) -> Union(List BSD,"failed")
         ++ brownianSD!([dB,...],dt) extended to \axiom{BSD} arguments
        brownianSD?:(List SDR,SDR) -> Boolean
         ++ brownianSD?([dB,...],dt) tests whether \axiom{[dB,...]}
         ++ is indeed a vector-valued Brownian motion in the
         ++ \axiom{dt} time-scale.
        brownianSD?:(List SDR,BSD) -> Boolean
         ++ brownianSD?([dB,...],dt) extended to \axiom{BSD} arguments
```
◇

File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

To begin with we consider construction of and test for classical differentials.

"ito2.spad" 39c ≡

```
     Implementation ==> add
      classicalSD!(dt:BSD):SDR ==
       alterQuadVar!(dt,dt,0::SDR)
       alterDrift!(dt,dt::SDR)
       for dZ in copyBSD() | not(dZ=dt) repeat
         alterQuadVar!(dt,dZ,0::SDR)
       dt::SDR

      classicalSD?(dt:SDR):Boolean ==
       (0 = dt**2) and (dt = drift dt)
      classicalSD?(dt:BSD):Boolean ==
       classicalSD?(dt::SDR)
```
◇

File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

As a test we introduce a basic time differential `dt`:

"ito2.input" 40a ≡

```
 -- ito2.input
 ⟨revision and author information 2a⟩
-- Description:
-- Further procedures for the ITO package.
⟨silence please 2c⟩
)library BSD SD SD2 ITO ITO2

dt := introduce!(t,dt) -- introduce time t
classicalSD!(dt)
classicalSD?(dt)
◇
```

File defined by scraps 40ac, 42, 44ab.

Next we need to be able to declare and test for a single Brownian motion, timed by another (classical) differential. Note that here we leave open the question of the relationship of the Brownian stochastic differential with other non-classical stochastic differentials!

"ito2.spad" 40b ≡

```

  brownianSD!(dB:BSD,dt:SDR):Union(SDR,"failed") ==
   not(classicalSD?(dt)) => "failed"
   alterQuadVar!(dB,dB,dt::SDR)
   for dZ in copyBSD() | classicalSD?(dZ::SDR)
     repeat alterQuadVar!(dB,dZ,0::SDR)
   alterDrift!(dB,0::SDR)
   dB::SDR
  brownianSD!(dB:BSD,dt:BSD):Union(SDR,"failed") ==
   brownianSD!(dB,dt::SDR)

  brownianSD?(dB:SDR,dt:SDR):Boolean ==
   (dt = dB**2) and (0 = drift dB)
  brownianSD?(dB:BSD,dt:BSD):Boolean ==
   brownianSD?(dB::SDR,dt::SDR)
◇
```

File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

To test this we introduce a Brownian motion `W`. Notice how we need to supply the correct timescale to verify the Brownian nature of the stochastic differential.

"ito2.input" 40c ≡

```
dW := introduce!(W,dW)
brownianSD!(dW,dt)
brownianSD?(dW,dt)
brownianSD?(dW,2*dt)
◇
```

File defined by scraps 40ac, 42, 44ab.

Now we want to introduce a further semimartingale `S` solving the (linear) stochastic differential equation

$$\mathrm{d}S \quad = \quad \mu S\,\mathrm{d}t \;+\; \sigma S\,\mathrm{d}W\,. \tag{5.1}$$

To do this we need a further procedure `itoSDE!`: however we first add a useful function `zeroSD?` which determines whether a stochastic differential can evaluate to zero because of second-order relations. Notice here the subtelty: we examine the second-order structure to determine whether the stochastic differential vanishes, because it is quite possible for a differential to vanish in effect *without* vanishing algebraically. (Indeed this begs the question of establishing links with Gröbner bases and algebraic geometry: we hope to return to this at another time.)

"ito2.spad" 41a ≡

```
zeroSD?(sd) ==
  ((0::SDR = drift sd)::Boolean) and ((0::SDR = sd**2)::Boolean)
```
◇

File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

Here are the procedures relating to specification using *sde*:

"ito2.spad" 41b ≡

```
itoSDE!(sde:Equation SDR):Union(SDR,"failed") ==
 dS := retract(lhs sde)@BSD
 dX := rhs sde
 not(freeOf?(dX,dS))            => "failed"
 alterDrift!(dS,drift dX)
 for dZ in copyBSD() | not(dZ=dS) repeat
  alterQuadVar!(dS,dZ,dX*(dZ::SDR))
 alterQuadVar!(dS,dS,dX*(dS::SDR))
 dS::SDR
itoSDE!(sde:Equation BSD):Union(SDR,"failed") ==
 itoSDE!((lhs sde)::SDR = (rhs sde)::SDR)

itoSDE?(sde:Equation SDR):Boolean == zeroSD?((lhs sde)-(rhs sde))
itoSDE?(sde:Equation BSD):Boolean ==
 itoSDE?((lhs sde)::SDR = (rhs sde)::SDR)
```
◇

File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

Next we need to be able to declare and test for vector-valued Brownian motions, timed by another (classical) differential, using the univariate version and the `uncorrelated!` code. The subsidiary procedures are forced on us by difficulties in compiling lists of basic stochastic differentials and by the need of the univariate `brownianSD!` to have defined quadratic variations for all other semimartingale differentials.

"ito2.spad" 41c ≡

```
brownianSD1!(ll:List BSD,dt:SDR):Boolean ==
 (0::Integer = #(ll)) => true
 dx := first ll
```

```
 alterQuadVar!(dx,dx,dt::SDR)$SDR
 (1::Integer = #(ll)) => true
 for dy in rest ll repeat
  alterQuadVar!(dx,dy,0::SDR)$SDR
 brownianSD1!(rest ll,dt)

brownianSD2!(ldB:List BSD,dt:SDR):Boolean ==
 (0::Integer = #(ldB)) => true
 brownianSD!(first ldB,dt) case "failed" => false
 brownianSD2!(rest ldB,dt)

brownianSD!(ll:List BSD,dt:SDR):Union(List BSD,"failed") ==
 not(classicalSD?(dt)) => "failed"
 brownianSD1!(ll,dt)
 not(brownianSD2!(ll,dt)) => "failed"
 ll

brownianSD!(ldB:List BSD,dt:BSD):Union(List BSD,"failed") ==
 brownianSD!(ldB,dt::SDR)

brownianSD?(ldB:List SDR,dt:SDR):Boolean ==
 not(uncorrelated?([[dB] for dB in ldB])) => false
 reduce("and",[brownianSD?(dB,dt) for dB in ldB],true)

brownianSD?(ldB:List SDR,dt:BSD):Boolean ==
 brownianSD?(ldB,dt::SDR)
```
◇

File defined by scraps 37, 38abc, 39abc, 40b, 41abc.

Here is how the itoSDE! code works out:

"ito2.input" 42 ≡

```
dS := introduce!(S,dS)
itoSDE!(dS = mu*S*dt + sg*S*dW)
itoSDE?(dS = mu*S*dt + sg*S*dW)
```
◇

File defined by scraps 40ac, 42, 44ab.

Example 5.1 : Notice we can solve this exactly! Consider the process Z, with

$$e^{(-q\ t+W\ p)}\,. \tag{5.2}$$

Then we can write in *AXIOM*:

```
Z := exp(p*W - q*t)
equation :=
    d Z = mu * Z * dt + sigma * Z * dW
zeroSD :=
    (lhs equation) - (rhs equation)
info := [coefficient(zeroSD,dt), coefficient(zeroSD,dW)]
```

which gathers the information together about p and q: in LaTeX output mode

AXIOM produces

$$\left[\frac{\left(-2\ q+p^2-2\ \mu\right)\ e^{\left(-q\ t+W\ p\right)}}{2},\ \left(-\sigma+p\right)\ e^{\left(-q\ t+W\ p\right)}\right]. \tag{5.3}$$

Type: List Expression Integer

We now must solve for `p` and `q`. Of course it is easy to see how to do it by hand; however it is a useful exercise to carry out the calculation in *AXIOM* so as to see how *AXIOM* tools might be used in more complicated calculations.

To carry out an algorithmic approach we need to remove the exponentials here, and then to convert the type to a list of polynomials. The first stage is easy in this case, though of course in general we would not be able to guarantee it.

```
info := [a/Z for a in info]
```

The second stage is slightly clumsy, as we need to pass from expressions to polynomials of fractions, and have to do this *via* fractions of polynomials (of course in this specific instance we could divide through by `Z/2` rather than by `Z`):

```
info := info::LIST FRAC POLY INT
info := info::LIST POLY FRAC INT
```

Finally we can apply the *AXIOM* solver from the `SystemSolvePackage`:

```
ans := solve(info,['p,'q])$SystemSolvePackage(INT)
```

This yields

$$\left[\left[p=\sigma,\ q=\frac{1}{2}\ \sigma^2-\mu\right]\right] \tag{5.4}$$

Type: List List Equation Polynomial Fraction Integer

(actually we used a coercion to force fractional coefficients for the polynomials).

Finally we can substitute back in `Z`:

```
subst(Z,first ans)
```

with result

$$e^{\frac{\left(-\sigma^2+2\ \mu\right)\ t+2\ W\ \sigma}{2}} \tag{5.5}$$

Type: Expression Integer

The answer is now readily checked: setting `ZZ := subst(Z,first ans)` we find that the *AXIOM* expression

```
(d ZZ) - (mu * ZZ * d t + sg * ZZ * d W)
```

simplifies to zero.

We now also introduce a bond B satisfying the simple (non-stochastic!) stochastic differential equation

$$\mathrm{d}B \quad = \quad \rho B \, \mathrm{d}t \tag{5.6}$$

using the following code:

"ito2.input" 44a ≡
```
dB :=introduce!(B,dB)
itoSDE!(dB = rho * B * dt)
itoSDE?(dB = rho * B * dt)
◇
```
File defined by scraps 40ac, 42, 44ab.

Example 5.2 : Notice again we can solve this exactly! It could be viewed as a special case of the previous exercise with $\sigma = 0$. In fact *AXIOM* already has procedures for solving such ordinary differential equations:

```
b := operator 'b
solve(D(b(t),t) - rho*b(t),b,t)
```

yields (in LaTeX form)

$$\left[particular = 0, \, basis = \left[e^{(\rho \ t)} \right] \right] \tag{5.7}$$

Finally consider an investment strategy V which is self-financing

"ito2.input" 44b ≡
```
V := operator 'V; smooth! V
-- we require a self-financing condition
-- V(S,t) = a*S + b*B

zero := V(S,t) - (a*S+b*B)

-- Solve the equations which arise by setting the following
-- equal to zero (raises issues about the algebra SDR,
-- such as zero-divisors!)
system := [ coefficient(drift d zero,dt), coefficient(dW*d zero,dt) ]
◇
```
File defined by scraps 40ac, 42, 44ab.

This should lead comfortably to the Black-Scholes formula of mathematical finance: see [1, 3] for good introductions on this subject.

```
ans := [ coefficient(drift d zero,dt), coefficient(dW*d zero,dt) ]
ans := ans::LIST FRAC DMP([a,b],EXPR INT)
...
```

Here is an example of the use of the multivariate Brownian code. First we build $n = 4$ independent Brownian motions A_1, A_2, A_3, A_4:

"sphere.input" 45a ≡

```
-- sphere.input
 ⟨revision and author information 2a⟩
-- Description: investigating BM on the n-sphere
⟨silence please 2c⟩
)library BSD SD SD2 ITO ITO2

dt := introduce!(t,dt) -- introduce time t
classicalSD!(dt)

n := 4
[introduce!(A[i],dA[i]) for i in 1..n]
brownianSD!([dA[i]::BSD for i in 1..n],dt)
brownianSD?([dA[i]::BSD::SD(INT) for i in 1..n],dt)
statusIto()
◇
```

File defined by scraps 45abc, 46abcd, 47.

Then we construct a Bessel process of dimension n and test it out.

"sphere.input" 45b ≡

```
R := sqrt(reduce(+,[A[i]^2 for i in 1..n]))

dW:=(d R)-drift d R; brownianSD?(dW,dt)
itoSDE?(d R = dW + ((n-1)/(2*R))*dt)
◇
```

File defined by scraps 45abc, 46abcd, 47.

The next step is to consider the B process normalized to lie on the unit sphere.

"sphere.input" 45c ≡

```
for i in 1..n repeat
 introduce!(U[i],dU[i])
 itoSDE!(dU[i]::BSD=d(A[i]/R))
◇
```

File defined by scraps 45abc, 46abcd, 47.

Note that `itoSDE()` is no longer useful at this stage, as it produces large quantities of indigestible output as soon as we start dealing with a substantial number of semimartingales with algegbraically non-trivial quadratic co-variations.

We study only the first coordinate to begin with, and we analyze it in a new time-scale. This is done by trial-and-error; clearly there is a place for a `DIFFUSE` package which uses algebra to test for when a semimartingale can be written in terms of a diffusion!

"sphere.input" 46a ≡

```
X := A[1]/R; dX := d X
dtau := dt/R^2
itoSDE?(dX^2 = (1-X^2)*dtau)
itoSDE?(drift d X = - ((n-1)/2)*X*dtau)
dB:=(d X - drift d X)/sqrt(1-X^2);brownianSD?(dB,dtau)
itoSDE?(d X = sqrt(1-X^2)*dB - ((n-1)/2)*X*dtau)
◇
```

File defined by scraps 45abc, 46abcd, 47.

The effect of all of this is to reveal X as solving the following stochastic differential equation, driven by a Brownian motion B in the $\mathrm{d}\tau$-time-scale:

$$\mathrm{d}X \quad = \quad \sqrt{1-X^2}\;\mathrm{d}B \;-\; \frac{n-1}{2}\,X\,\mathrm{d}\tau \tag{5.8}$$

We now must consider the behaviour of the remaining components, namely `[dU[i] for i in 2..n]`. What happens when we re-normalize these to lie on a unit sphere? Consider the second coordinate: we verify something which actually follows easily from symmetry considerations; it satisfies a similar stochastic differential equation!

"sphere.input" 46b ≡

```
S := sqrt(reduce(+,[A[i]^2 for i in 2..n]))
Y := A[2]/S; dY := d Y
dsigma := dt/S^2
dC:=(d Y - drift d Y)/sqrt(1-Y^2);brownianSD?(dC,dsigma)
itoSDE?(d Y = sqrt(1-Y^2)*dC - ((n-2)/2)*Y*dsigma)
◇
```

File defined by scraps 45abc, 46abcd, 47.

The skew-product nature of this construction is revealed if we note the following equation holds:

"sphere.input" 46c ≡

```
itoSDE?(dsigma=dtau*(R/S)^2)
◇
```

File defined by scraps 45abc, 46abcd, 47.

As a final step, we consider the relationship of the three-dimensional case Y to Legendre polynomials. First we define the Legendre polynomials:

"sphere.input" 46d ≡

```
p(0,x) == 1
p(1,x) == x
p(n,x) == ((2*n-1)*x*p(n-1,x) - (n-1)*p(n-2,x))/n
◇
```

File defined by scraps 45abc, 46abcd, 47.

We then find the Legendre polynomials do indeed give us spherical harmonics (of the first kind):

"sphere.input" 47 ≡

```
for m in 1..10 repeat
 print(itoSDE?(
  drift d p(m,Y) = -(m*(m+1)/2) * p(m,Y) * dsigma
   ))$OutputForm
```

◇

File defined by scraps 45abc, 46abcd, 47.

Chapter 6

Package for Itô integrals

We add a package to implement the Itô integral.

"itointegral.spad" 49a ≡

```
-- itointegral.spad
⟨revision and author information 2a⟩
)abbrev package ITOINT ItoIntegral
--% ItoIntegral
-- Compilation order:
-- BSD SD SD2 ITO ITO2 ITOINT OLDITO
--                     ******
++ Basic Operations: smooth!, itoIntegral
++ Related Domains: BasicStochasticDifferential,
++                  StochasticDifferential
++ Also See: Ito, Ito2
++ AMS Classifications:
++ Keywords: stochastic differential, semimartingale.
++ Examples:
++ References:
++ Description:
++ Implements the Ito integral
++
ItoIntegral(R:Join(OrderedSet, IntegralDomain)):
 Exports == Implementation where
  BOP  ==> BasicOperator
  EXR  ==> Expression(R)
  FR   ==> Fraction(Polynomial(R))
  SDR  ==> StochasticDifferential(R)
◇
```

File defined by scraps 49ab, 50ab.

There is just one export, namely `itoIntegral`:

"itointegral.spad" 49b ≡

```
  Exports ==> with
   itoIntegral:(EXR,EXR)->EXR
    ++ itoIntegral(X,Y) stands for the Ito integral of \axiom{X}
    ++ with respect to \axiom{dY}
   itoIntegral:(FR,FR)->EXR
    ++ itoIntegral(X,Y) stands for the Ito integral of \axiom{X}
```

```
  ++ with respect to \axiom{dY}
 itoIntegral:SDR -> EXR
  ++ itoIntegral(sd) is converted into a sum of elements of the
  ++ form \axiom{itoIntegral(X,Y)}
◇
```

File defined by scraps 49ab, 50ab.

We need an operator `itoIntegral`. In kernel form this appears as

```
itoIntegral(X,Y)
```

to represent $\int X\,\mathrm{d}Y$. We conduct a very elementary simplification:

"itointegral.spad" 50a ≡

```
Implementation ==> add
 opitoIntegral := operator('itoIntegral,2)$BOP

 itoIntegral(X:EXR,Y:EXR):EXR ==
  itoIntegral((X*d(Y)$Ito(R))::SDR)::EXR

 itoIntegral(X:FR,Y:FR):EXR == itoIntegral(X::EXR,Y::EXR)
◇
```

File defined by scraps 49ab, 50ab.

The single-argument form, for which the argument is a stochastic differential, converts to a linear combination of kernel forms, ensuring that the second argument is directly related to a basic stochastic differential:

"itointegral.spad" 50b ≡

```

 itoIntegral(sd:SDR):EXR ==
  lsd  := listSD(sd)
  lx := [coefficient(sd,ds)::EXR for ds in lsd]
  ly := [getSmgl(ds)::Symbol::EXR for ds in lsd]
  l := [opitoIntegral(x,y) for x in lx for y in ly]
  reduce("+",l,0)
◇
```

File defined by scraps 49ab, 50ab.

Finally, here is a test of `itoIntegral`:

"itointegral.input" 50c ≡

```
-- itointegral.input
⟨revision and author information 2a⟩
-- Description:
-- Test of the ITOINT package.
⟨silence please 2c⟩
)library BSD SD SD2 ITO ITO2 ITOINT
◇
```

File defined by scraps 50c, 51abcd.

We first introduce semimartingales for testing purposes "itointegral.input" 51a ≡

```
dt:=introduce!(t,dt);  classicalSD!(dt);
dB:=introduce!(B,dB);  brownianSD!(dB,a11*dt);
dC:=introduce!(C,dC);  brownianSD!(dC,a22*dt);
alterDrift!(dB,a1*dt); alterDrift!(dC,a2*dt);
alterQuadVar!(dB,dC,a12*dt);
```
◇

File defined by scraps 50c, 51abcd.

Here are some simple examples of itoIntegral in action:

"itointegral.input" 51b ≡

```
itoIntegral(B,C)
itoIntegral(B+C,C)
itoIntegral(B,C+C^2)
itoIntegral(B,C+C^2*B)
```
◇

File defined by scraps 50c, 51abcd.

Here are some more involved examples:

"itointegral.input" 51c ≡

```
x:=B; y:=C+C^2;   d(itoIntegral(x,y)) - x*d(y)
x:=B; y:=C+C^2*B; d(itoIntegral(x,y)) - x*d(y)
x:=sin(B); y:=cos(C); d(itoIntegral(x,y)) - x*d(y)
x:=sin(B)+c;  y:=1+cos(B)^2+sin(sin(B))*C;
d(itoIntegral(x,y)) - x*d(y)
```
◇

File defined by scraps 50c, 51abcd.

Note that itoIntegral behaves well with respect to freeOf? and also works well under iteration:

"itointegral.input" 51d ≡

```
freeOf?(itoIntegral(B,C)*dC,dB)

h:=itoIntegral(itoIntegral(B*dC)*dC)
d h
```
◇

File defined by scraps 50c, 51abcd.

Chapter 7

Derivation of the Mardia-Dryden shape distribution

In this chapter we summarize how to use the *AXIOM* version of symbolic Itô calculus to derive the Mardia-Dryden distribution from shape theory. Full mathematical details are given in [11]: here we concentrate on the computer algebra story! We begin by entering *AXIOM* and reading in the *Itovsn3* script using

```
)read MardiaDryden.input
```

The script begins by loading the *Itovsn3* libraries:

"MardiaDryden.input" 53a ≡

```
-- MardiaDryden.input
⟨revision and author information 2a⟩
-- Description:
-- Demonstration of derivation of Mardia-Dryden distribution using
-- stochastic calculus
⟨silence please 2c⟩
)library BSD SD SD2 ITO ITO2
◇
```

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

We begin by setting up the spatial dimension `n=2`, the number of points `pts=3`, and the fundamental time semimartingale:

"MardiaDryden.input" 53b ≡

```
n:= 2; pts := 3;                           -- Bureaucracy

dt := introduce!(t,dt)                     -- Time parameter
classicalSD!(dt)
◇
```

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

The size process uses an associated dimension `n*(pts-1)`, and is a Bessel process driven by a specified Brownian motion `A`. From the size process we derive a time-change differential dtau:

"MardiaDryden.input" 54a ≡
```
dim := n*(pts-1);

dA := introduce!(A,dA);                        -- Brownian noise
brownianSD!(dA,dt);
dR := introduce!(R,dR)                         -- Bessel process
itoSDE!(dR = dA + ((dim-1)/(2*R))*dt);
dtau := (dt/R^2)                               -- Time change
```
◇

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

We now need a Brownian innovation process, working with the time-change differential but uncorrelated to it:

"MardiaDryden.input" 54b ≡
```

dW := introduce!(W,dW)                         -- BM (size innovation)
brownianSD!(dW,dtau)
uncorrelated!([dA,dR],[dW])                    -- uncorrelated with size
```
◇

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

The distance of the shape from the equilateral triangle shape point is a "Bessel process on the sphere":

"MardiaDryden.input" 54c ≡
```

dS := introduce!(S,dS)                         -- Bessel process on S2
itoSDE!(dS = dW + (cos(2*S)/sin(2*S))*dtau) -- need cancellations
statusIto()                                    -- of  cos/sin
```
◇

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

We now need to build up the "Kolmogorov backwards partial differential equation" for the transition density. Notice this terminology is not strictly accurate! we are working with a process rendered non-Markovian by the time-change. However the required density still becomes a martingale when its first argument is a shape-diffusion. We use *AXIOM* "scripting" capability to work with a more elegant t_1 rather than t1.

"MardiaDryden.input" 54d ≡
```
t1 := subscript(t,[1])
kappa := R^2/(4*(t1-t))                        -- parameter
kappaSub := first solve(kappa=k,t1)            -- useful substitution

-- derive backwards differential equation
f := operator 'f; smooth!(f);
density := f(cos(2*S),kappa)                   -- density
bde := coefficient(drift d density,dt)         -- backwards DE
bde :=
  0 = numer (subst(bde,kappaSub)/2)
```
◇

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

$$
\begin{aligned}
0 \quad = \quad & k^2\ f_{,2,2}\left(\cos\left(2\ S\right),\ k\right) + \sin\left(2\ S\right)^2\ f_{,1,1}\left(\cos\left(2\ S\right),\ k\right) \\
& +\left(2\ k^2 + 2\ k\right)\ f_{,2}\left(\cos\left(2\ S\right),\ k\right) - 2\ \cos\left(2\ S\right)\ f_{,1}\left(\cos\left(2\ S\right),\ k\right)
\end{aligned}
\tag{7.1}
$$

The backwards differential equation is not very illuminating. We return to the semimartingale picture and try a trial possibility for the density, based on perturbations around a "flat-space" version, itself suggested by the geometry (shape space is $\Sigma_2^3 = S^2(\frac{1}{2})$). Again we use the scripting facility for the sake of cosmetic elegance!

"MardiaDryden.input" 55a ≡

```
p0 := kappa*exp(-kappa*(1-cos(2*S)))

a0 := subscript(a,[0])
a1 := subscript(a,[1])
a2 := subscript(a,[2])

p1 := (a0+a1/kappa+a2*(1-cos(2*S)))*p0
◇
```

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

We will perturb around time-zero, space-zero so add this to our previous simplifying substitution:

"MardiaDryden.input" 55b ≡

```
reference := [S=0,kappaSub]
◇
```

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

First adduce the condition that the drift is zero at time zero, space zero

"MardiaDryden.input" 55c ≡

```
zero := coefficient(drift d p1,dt)/p0
zeroDrift :=
  0 = simplify subst(zero,reference)
◇
```

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

yielding the following:

$$
\frac{4\ a_2 - 4\ a_1 + 4\ a_0}{R^2} \quad = \quad 0\,. \tag{7.2}
$$

Now consider the requirement that the density integrates up to unit mass on $S^2(\frac{1}{2})$:

"MardiaDryden.input" 55d ≡

```
mass := integrate(subst(p1,kappaSub)*sin(2*S),S)
mass := simplify(subst(mass,S=%pi/2)-subst(mass,S=0))
unitMass := mass=1
◇
```

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

This yields

$$
\frac{\left(\left(-2\ a_2 - a_0\right)\ k - a_2 - a_1\right)\ e^{\left(-2\ k\right)} + a_0\ k + a_2 + a_1}{2\ k} \quad = \quad 1\,. \tag{7.3}
$$

But now observe that $a0$, $a1$, $a2$ are constants. This means they cannot vary with k!

"MardiaDryden.input" 56a ≡
```
constancy := subst(D(2*k*unitMass,k),k=0)
```
◇

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

This delivers

$$(2\ a_1 - a_0)\ e^0 + a_0 \quad = \quad 2\,. \tag{7.4}$$

and we can use the resulting system of equations to obtain a solution density

"MardiaDryden.input" 56b ≡
```
solution := first solve([constancy, unitMass, zeroDrift],[a0,a1,a2])
```
◇

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

and hence a candidate for the Mardia-Dryden density:

"MardiaDryden.input" 56c ≡
```
subst(mddensity:=subst(p1,solution),kappaSub)
```
◇

File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

$$(k\ \cos(2\ S) + k + 1)\ e^{(k\ \cos(2\ S) - k)}\,. \tag{7.5}$$

Various tests now show this works! We can check on the effects of the constraints we imposed. *A priori* the constraints should only hold to second order at time-zero, space-zero. However they actually do much better than that:

- we obtain 0 for the drift *for all time*:

  "MardiaDryden.input" 56d ≡
  ```
  subst(coefficient(drift d mddensity,dt),
      sin(2*S)=sqrt(1-cos(2*S)^2))
  ```
  ◇

  File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

  and this of course means that the "Kolmogorov backwards partial differential equation" for the transition density is solved by `mddensity`.

- unit total mass *for all time*, checked by integrating against the density appropriate for Σ_2^3:

  "MardiaDryden.input" 56e ≡
  ```
  ii := integrate(mddensity*sin(2*S),S)
  simplify(subst(ii,S=%pi/2) - subst(ii,S=0))
  ```
  ◇

  File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

- and (something which *doesn't* come straight from our constraints!) appropriate behaviour for t near t1: the limit below is finite and non-zero!

```
"MardiaDryden.input" 57a ≡
      asympt := mddensity/p0
      limit(asympt,t=t1)
      ◇
```
File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

- For completeness we finally show how to check that the density really does satisfy the "Kolmogorov backwards partial differential equation":

```
"MardiaDryden.input" 57b ≡
      bde
      mdd := subst(mddensity,[cos(2*S)=c,kappaSub])
      subst(bde,cos(2*S)=c)
      (rule f(c,k) == mdd) rhs subst(bde,cos(2*S)=c)
      ◇
```
File defined by scraps 53ab, 54abcd, 55abcd, 56abcde, 57ab.

  and it will be seen that the answer is `0`.

We deduce that Eq. (7.5) really is the Mardia-Dryden density.

Appendix A: Older version of the Itô formula package

This is a very simple implementation of the Itô differential, useful for testing purposes. It is based on differentiation with respect to constituent basic semimartingales. This has the strong disadvantages of not working when the target expression contains `itoIntegral` terms, and of assuming appropriate differentiability by default. In earlier versions of *Itovsn3* (*REDUCE* and *Mathematica* implementations) we worked round some of these problems, for example by adding special differentiation rules for `itoIntegral`. In the *AXIOM* version we have implemented `d` in a more fundamental way: we retain the simple implementation below (`OldD`) to help test the behaviour of `d`.

"oldito.spad" 59 ≡

```
-- oldito.spad
⟨revision and author information 2a⟩
)abbrev package OLDITO OldIto
--% OldIto
-- Compilation order:
-- BSD SD SD2 ITO ITO2 ITOINT OLDITO
--                          ******
++ Basic Operations: OldD
++ Related Domains: BasicStochasticDifferential,
++                  StochasticDifferential
++ Also See:
++ AMS Classifications:
++ Keywords: stochastic differential, semimartingale.
++ Examples:
++ References:
++ Many! But also see Kendall (1992) for a geometric treatment based
++ on second-order approximation.
++ Description:
++ Implements the stochastic differential.
++
OldIto(R:Join(OrderedSet, IntegralDomain)): Exports == Implementation where
 EX ==> Expression R
 PX ==> Polynomial R
 FX ==> Fraction Polynomial R
 BSD ==> BasicStochasticDifferential
 SDR ==> StochasticDifferential(R)
◇
```

File defined by scraps 59, 60ab.

Note that we need several variants of `OldD` in order to capture all the appropriate domains.

"oldito.spad" 60a ≡

```
       Exports ==> with
        OldD:EX -> SDR
         ++ OldD(X) yields the stochastic differential of \axiom{X}
        OldD:FX -> SDR
         ++ OldD(X) extended to \axiom{Fraction Polynomial R}
```

◇

File defined by scraps 59, 60ab.

"oldito.spad" 60b ≡

```
       Implementation ==> add

        OldD(Y:EX):SDR ==
         0 = height Y => 0$SDR
         smgls := [ k for k in variables Y | d(k)$BSD case BSD ]
         reduce("+",[(D(Y,s)*(((d$BSD)s)::BSD::SDR)) for s in smgls], 0$SDR)
          + ((1::EX/2::EX))* reduce("+",[ reduce("+",[
           (D(Y,[r,s])*(((d$BSD)r)::BSD::SDR)*(((d$BSD)s)::BSD::SDR))
             for r in smgls],0$SDR) for s in smgls],0$SDR)

        OldD(Y:FX):SDR == OldD(Y::EX)
```

◇

File defined by scraps 59, 60ab.

Appendix B: Future work

In a later version we will consider whether we need a `Semimartingale` domain. There is also a question whether one should implement classical differentials as forming a sub-domain $\mathcal{D}_1$ of $\mathcal{D}_2$.

It appears that the comment on Page 41 can be developed into a kind of algebraic geometry on the module of second-order differentials $\mathcal{D}_2$.

An automatic procedure for solving linear *sde* would be useful (implementing a "stochastic Liouville theorem" and more: see Kunita's work [13, § 4.9], also [15]) and therefore will be implemented at a later stage (indeed this has already been done in the *REDUCE* version of *Itovsn3*). While algorithmically trivial, such a procedure can be of enormous use in dealing with linear *sde* whose coefficients are nonetheless of complicated parametric form.

Similarly one should develop a notion of "automatic diffusion geometry" as in the *REDUCE* and *Mathematica* versions of *Itovsn3*. This should undertake several tasks:

- detecting when a semimartingale can be written as a diffusion (with suitable restrictions on the coefficients, and perhaps taking care to use the correct time-scale);
- carrying out a formal Feller-diffusion analysis;
- computing intrinsic drift and Riemannian geometry for a multidimensional elliptic diffusion.

Some efficiency gains are possible for `d`: note the possible efficiency gain if we were to test for vanishing of the square of `sd`. Note too that a more sophisticated version of the `"ito"` property might pre-compute derivatives .... However both these options would increase code-complexity.

Finally it is planned to construct some simple integration procedures for `itoIntegral`.

Bibliography

[1] M. Baxter and A. Rennie.
Financial Calculus: an introduction to derivative pricing.
Cambridge University Press, Cambridge, 1996.

[2] P. Briggs.
`Nuweb` Version 0.87b: A simple literate programming tool.
Published on the World-Wide Web by `preston@cs.rice.edu`, 1992.

[3] D.J. Hand and S.D. Jacka.
Statistics in Finance.
Arnold, London, 1998.

[4] K. Itô.
Stochastic differentials.
Applied Mathematics and Optimization, 1:374–381, 1975.

[5] W.S. Kendall.
Symbolic Itô calculus: An introduction.
Research Report 217, Department of Statistics, University of Warwick, 1991.

[6] W.S. Kendall.
Symbolic Itô calculus: An overview.
In N. Bouleau and D. Talay, editors, *Probabilités Numeriques*, pages 186–192, Rocquencourt, 1991. INRIA.

[7] W.S. Kendall.
A remark on the proof of Itô's formula for C^2 functions of continuous semimartingales.
Journal of Applied Probability, 29:216–221, 1992.

[8] W.S. Kendall.
Computer algebra in probability and statistics.
Statistica Neerlandica, 47:9–25, 1993.

[9] W.S. Kendall.
Doing stochastic calculus with *Mathematica.*
In H. Varian, editor, *Economic and Financial Modeling with* Mathematica, pages 214–238. Springer-Verlag, New York, 1993.

[10] W.S. Kendall.
A diffusion model for Bookstein triangle shape.
Advances in Applied Probability, 30(2):317–334, 1998.

[11] W.S. Kendall.

Symbolic Itô calculus: an ongoing story.
Research Report 327, Department of Statistics, University of Warwick, 1998.
Accepted for publication in *Statistics and Computing* subject to revision.

[12] D.E. Knuth.
Literate programming.
The Computer Journal, 27(2):97–111, May 1984.

[13] H. Kunita.
Stochastic flows and stochastic differential equations.
Cambridge University Press, 1990.

[14] Silvio Levy and Donald E. Knuth.
`CWEB` user manual: The CWEB system of structured documentation.
Technical Report STAN-CS-83-977, Stanford University, October 1990.
Available for anonymous ftp from `labrea.stanford.edu` in directory `pub/cweb`.

[15] Y. Yamato.
Stochastic differential equations and nilpotent Lie algebras.
Zeitschrift für Wahrscheinlichkeitstheorie und Verve Gebiete, 47:213–229, 1979.

Other University of Warwick Department of Statistics Research Reports authored or co–authored by W.S. Kendall.

161: The Euclidean diffusion of shape.
162: Probability, convexity, and harmonic maps with small image I: Uniqueness and fine existence.
172: A spatial Markov property for nearest–neighbour Markov point processes.
181: Convexity and the hemisphere.
202: A remark on the proof of Itô's formula for C^2 functions of continuous semimartingales.
203: Computer algebra and stochastic calculus.
212: Convex geometry and nonconfluent Γ-martingales I: Tightness and strict convexity.
213: The Propeller: a counterexample to a conjectured criterion for the existence of certain convex functions.
214: Convex Geometry and nonconfluent Γ-martingales II: Well–posedness and Γ-martingale convergence.
216: *(with E. Hsu)* Limiting angle of Brownian motion in certain two–dimensional Cartan–Hadamard manifolds.
217: Symbolic Itô calculus: an introduction.
218: *(with H. Huang)* Correction note to "Martingales on manifolds and harmonic maps."
222: *(with O.E. Barndorff-Nielsen and P.E. Jupp)* Stochastic calculus, statistical asymptotics, Taylor strings and phyla.
223: Symbolic Itô calculus: an overview.
231: The radial part of a Γ-martingale and a non-implosion theorem.
236: Computer algebra in probability and statistics.
237: Computer algebra and yoke geometry I: When is an expression a tensor?
238: *Itovsn3*: doing stochastic calculus with *Mathematica*.
239: On the empty cells of Poisson histograms.
244: *(with M. Cranston and P. March)* The radial part of Brownian motion II: Its life and times on the cut locus.
247: Brownian motion and computer algebra (Text of talk to BAAS Science Festival '92, Southampton Wednesday 26 August 1992, with screenshots of illustrative animations).
257: Brownian motion and partial differential equations: from the heat equation to harmonic maps (Special invited lecture, 49th session of the ISI, Firenze).
260: Probability, convexity, and harmonic maps II: Smoothness via probabilistic gradient inequalities.
261: *(with G. Ben Arous and M. Cranston)* Coupling constructions for hypoelliptic diffusions: Two examples.
280: *(with M. Cranston and Yu. Kifer)* Gromov's hyperbolicity and Picard's little theorem for harmonic maps.
292: Perfect Simulation for the Area-Interaction Point Process.
293: *(with A.J. Baddeley and M.N.M. van Lieshout)* Quermass-interaction processes.
295: On some weighted Boolean models.
296: A diffusion model for Bookstein triangle shape.
301: COMPUTER ALGEBRA: an encyclopaedia article.

308: Perfect Simulation for Spatial Point Processes.
319: Geometry, statistics, and shape.
321: From Stochastic Parallel Transport to Harmonic Maps.
323: *(with E. Thönnes)* Perfect Simulation in Stochastic Geometry.
325: *(with J.M. Corcuera)* Riemannian barycentres and geodesic convexity.
327: Symbolic Itô calculus in *AXIOM*: an ongoing story.
328: *Itovsn3* in *AXIOM*: modules, algebras and stochastic differentials.
331: *(with K. Burdzy)* Efficient Markovian couplings: examples and counterexamples.
333: Stochastic calculus in *Mathematica*: software and examples.
341: Stationary countable dense random sets.
347: *(with J. Møller)* Perfect Metropolis-Hastings simulation of locally stable point processes.
348: *(with J. Møller)* Perfect implementation of a Metropolis-Hastings simulation of Markov point processes

Also see the following related preprints

317: E. Thönnes: Perfect Simulation of some point processes for the impatient user.
334: M.N.M. van Lieshout and E. Thönnes: A Comparative Study on the Power of van Lieshout and Baddeley's J-function.

If you want copies of any of these reports then please mail your requests to the secretary of the Department of Statistics, University of Warwick, Coventry CV4 7AL, UK.