Introduction

This worksheet contains a Maple implementation of the Secure Hash Algorithm-3 (SHA-3) family of functions wich have been standardized by the US National Institute of Standards and Technology (NIST) in August 2015, as specified in [FIPS PUB 202] (SHA-3 Standard). The SHA-3 family consists of four cryptographic hash functions, called SHA3-224, SHA3-256, SHA3-384, and SHA3-512, and two extendable-output functions (XOFs), called SHAKE128 and SHAKE256. The XOFs are different from hash functions but, as stated in SHA-3 Standard, "it is possible to use them in similar ways, with the flexibility to be adapted directly to the requirements of individual applications, subject to additional security considerations". The SHA-3 functions are based on the Keccak sponge function, designed by G. Bertoni, J. Daemen, M. Peeters, G. Van Assche. Keccak was selected for this purpose because it was declared, on October 2, 2012, the winner of the NIST Hash Function Competition held by NIST.

The worksheet is an updated version of a previous one which had been published in 2013 with the title "The SHA-3 family of hash functions and their use for message authentication". The new version includes the XOFs following the NIST specification and, at the programming level, the most important change is in the padding procedure. In the original proposal, the SHA-3 functions were instances of Keccak, which uses the pad10*1 algorithm specified in Kreference, but in SHA-3 Standard the application of Keccak is preceded by appending a two-bit suffix "01" to the message in the case of the SHA-3 hash functions and appending a four-bit "1111" suffix in the case of the SHA-3 XOFs. The stated purpose of this modification is to distinguish the SHA-3 hash functions from the XOFs and to facilitate the development of new variants of the SHA-3 functions. With respect to the previous version of this worksheet, we just made the minimum number of changes to comply with the standard and, in particular, we kept the terminology used in the previous version. We also implement the Message Authentication Code HMAC-SHA-3, based on the SHA-3 hash functions.
In addition to the NIST specification in SHA-3 Standard, the principal reference for Keccak may be found in the The Keccak sponge function family web page and, in particular, in the documents "The Keccak reference" (Reference), "Keccak implementation overview" (Kimplementation) and, in more detail, in "Keccak sponge function family main document" (Kmain). The implementation in this worksheet is similar to other implementations of cryptographic schemes included in [Gómez Pardo] and its purpose is to facilitate learning through experimentation and to help to understand how the algorithm works. With this in mind, one important design goal was to build up an implementation that could handle real messages in a variety of formats and not just short text strings. The implementation uses, whenever possible, lookup tables for efficiency and is able to compute hashes or authentication tags for both text (or hex) strings and binary files of moderate size.

A hash function maps messages (which may be regarded as bit strings of arbitrary length) to short fixed-length strings, the message digests or hash values (or, simply, hashes) which can be seen as a sort of fingerprints of the messages. The idea is then that any change, either accidental or intentional to the message will, with high probability, also change the hash value, allowing the detection of the fact that these changes have taken place. For the hash function to be cryptographically useful it is then a basic requirement that the function be collision resistant, in the sense that it should be infeasible for an adversary to generate a collision, i.e., two different messages with the same hash (since hashes have a short fixed length, there will be many messages which have the same hash value, so the hash function is far from being injective but it should look like that to an adversary). At first sight, it might seem sufficient that the adversary be unable to find a collision with a given message, but for most cryptographic purposes this is not enough and, on the other hand, finding collisions between two messages, none of which is fixed, is much easier as a consequence of the birthday attack (see Wikipedia Cryptographic Hash Function and [Gómez Pardo, 5.6]). Thus a cryptographic hash function should be collision resistant (meaning that finding a collision is infeasible in practice), and also preimage resistant (i.e., one-way), the latter being always the case for all genuine collision resistant functions (i.e., for those that are not specifically designed to behave differently). Cryptographic hash functions have many important cryptographic applications, including password or pin verification (the passwords are not stored but their hashes are stored instead), digital signatures and message authentication through MACs (our implementation of HMAC-SHA-3 below is an application of the latter type). In the case of the four SHA-3 hash functions, SHA3-224, SHA3-256, SHA3-384, and SHA3-512, the suffix after the dash indicates the bit-length of the digest, so that, for example, SHA3-512 produces 512-bit hashes. An extendable-output function (abbreviately XOF) maps messages to bit strings but, in contrast with hash functions, its output can be extended to any desired length. Thus in the functions SHAKE128 and SHAKE256, the suffixes do not indicate output length but, as explained in SHA-3 Standard, they refer to the security strength that these functions can generally support.

The Keccak algorithm is based on the sponge construction, which is more general than a hash function since, using a permutation of fixed-length bit strings, it allows the selection of an arbitrary output length; this construction is presented in the documents "Cryptographic Sponge Functions" (Csponge) and the already mentioned Kmain, where its security properties are discussed. In particular, the authors have shown that the Keccak sponge function has the property called indifferentiability from a random oracle (a random oracle is an ideal hash function which, on each new input query, will pick uniformly at random some response from its output domain and will always return the same response if asked the same query again). Indeed, the authors have obtained an upper bound for the expected success probability of a shortcut attack (an attack whose success probability against the sponge function is higher than against a random oracle) which, in turn, gives a lower bound for the expected complexity of differentiating the sponge construction (calling a random permutation) from a random oracle. This is a provable lower bound on the expected workload of any successful generic attack, namely, of an attack that does not exploit specific properties of the underlying permutation, cf. Kmain. In addition to these good security properties, the
sponge construction also has other practical advantages over constructions that make use of a compression function (a hash function that acts on fixed-length messages) such as the Merkle-Damgård construction used in the SHA-2 family of hash functions (see, for example, [Gómez Pardo, 5.6.2] for a description of this construction). Among these advantages are its greater simplicity, the fact that it has variable-length output, and also the flexibility provided by the fact that its security level can be increased at the cost of speed by decreasing the bit rate and increasing the capacity (see below for the definition of these parameters).

The sponge construction builds a function \( \text{SPONGE}[f, \text{pad}, r] \) which maps arbitrary finite-length bit strings to infinite-length bit strings using a permutation \( f: (\mathbb{Z}_2)^b \rightarrow (\mathbb{Z}_2)^b \) (where the positive integer \( b \) is called the width of \( f \)), an appropriate padding rule "pad" and a parameter \( r \) (a positive integer < \( b \)) called the bit rate. A finite-length output can be obtained by truncating it to its first \( n \) bits. The sponge construction has a state of \( b \) bits which at the start of the process are all initialized to 0. The input message is then padded and divided into \( r \)-bit blocks and the sponge construction proceeds in two phases called the absorbing phase and the squeezing phase. The state bits are separated into two parts: the outer part consisting of the first \( r \) bits and the inner part with the remaining \( c = b - r \) bits, where \( c \) is called the capacity.

In the absorbing phase, the first message block is XORed (using the bitwise XOR) with the outer part of the initial 0-state and then the permutation \( f \) is applied to the resulting state. Then the next \( r \)-bit message block is XORed with the outer part of the state and \( f \) is applied again. This process continues interleaving the XOR operations with applications of the permutation \( f \) until all the input is consumed. When all the message blocks are processed, the sponge construction switches to the squeezing phase. This phase starts by taking the first \( r \) bits from the state and then applying the permutation \( f \). If the squeezed \( r \) bits are less than the requested \( n \) bits, the process continues by applying \( f \) to the state and then extracting the first \( r \) bits of the resulting state. The process is continued interleaving the extraction of the first \( r \) bits of the state with the application of \( f \) until the total number of bits obtained is \( n \), and the final output is obtained by truncating the squeezed bits to the first \( n \) bits. The whole process is summarized in \( \text{Cspponge} \) by means of the following image:

![Image of the sponge construction process](image)

The Keccak sponge construction is the special case of this algorithm based on the use of the Keccak permutations \( \text{Keccak}_f[b] \) (which play the role of the permutation \( f \) above), together with multi-rate padding, described below. As mentioned in \( \text{Kreference} \), the design philosophy
underlying the Keccak construction is the so-called hermetic sponge strategy, consisting in the use of the sponge construction for obtaining provable security against generic attacks and calling a permutation that should not have structural properties with the exception of a compact description; this philosophy is explained in detail in Csponge and, more specifically for the case of the Keccak permutations, in Kmain. There are seven Keccak permutations Keccak-f[b]: (\(\mathbb{Z}_2\))\(^b\) \rightarrow (\(\mathbb{Z}_2\))\(^b\), of widths \(b\in\{25, 50, 100, 200, 400, 800, 1600\}\), but here we will consider only the values of \(b\) which are multiples of 8, i.e., 200, 400, 800 and 1600. The reason is that our implementation will be byte-oriented and so all the bit strings or bit lists we will handle will have length a multiple of 8. This is not a very serious restriction because the usual messages have this property and, on the other hand, the lower values of \(b\) are weak from the security point of view and, in fact, the only value used in the SHA-3 standard is \(b = 1600\). The permutation width \(b\) can be written in the form \(b = 25 \times 2^\ell\), for \(\ell\) ranging from 3 to 6.

The permutation Keccak-f[b] is described in Kreference as a sequence of operations on a state \(a\) that is a three-dimensional array of elements of \(\mathbb{Z}_2\) of the form \(a[5][5][w]\), with \(w = 2^\ell\). Here, a bit string \(s\in(\mathbb{Z}_2)^b\) whose bits we assume enumerated from 0 to \(b-1\), is mapped to the state by assigning the bit \(s[w(5y+x)+z]\) to the state bit \(a[x][y][z]\), and the state bits \(a[x][y][0..w-1]\), for fixed values of \(x\) and \(y\), constitute a lane, so that the state can be viewed as a \(5 \times 5\) array of lanes. We refer to Kreference for the description of Keccak-f[b], which is an iterated permutation consisting of a sequence of \(n_r\) rounds, where \(n_r = 12 + 2\ell\) and each round consists of five steps (Kreference). As described above, the permutation width \(b\) is written in the form \(b = c + r\), where \(c\) is the capacity and \(r\) is the bit rate. The message is padded to make its length a multiple of \(r\) and the blocks of length \(r\) are successively absorbed into the state, each of these absorptions being followed for an application to the state of the Keccak-f[b] permutation. Although this is not necessary, we will for simplicity assume that \(c\) (and hence also \(r\)) is a multiple of 8, and, in fact, we will require that \(r\) be a multiple of \(w\), which is always the case for the SHA-3 hash functions included in the SHA-3 standard (see SHA-3 Standard). In the original proposal, Ksubmission, it is suggested, for security reasons, the use of \(c = 2n\), where \(n\) is the size of the output hash and this suggestion has been followed in the NIST standard. Thus \(r\), the number of message bits processed per block permutation, depends on the output hash size. This way the rate \(r = 1152, 1088, 832, 576\), respectively, for 224, 256, 384 and 512-bit hash sizes when \(w\) is 64, which corresponds to \(b = 1600\). As far as possible, we will keep these notations in our implementation. In the NIST specification in SHA-3 Standard, a more general version of the Keccak family of permutations is described and, in particular, the Keccak-f[b] permutation is equivalent to NIST’s Keccak-p[b, 12+2\ell] so that Keccak-p[1600, 24], which underlies the six SHA-3 functions defined in the NIST standard, is equivalent to Keccak-f[1600] in the terminology of Kreference that we are using here.

There are many possible choices for working with the data involved in the algorithm and, in particular, the lanes in the state might be represented by decimal integers, by hex strings, by lists of bytes or by lists (or strings) of bits. In this implementation we use integers to represent the lanes and the bitwise operations on them in the Keccak rounds are performed with the help of Maple's package Bits, with the exception of the cyclic bit rotation, for which we use the function Rot defined below. When converting between bit lists and integers, we will use the little endian convention as suggested in Kreference, so that bits will be enumerated in ascending order from the least significant to the most significant one and the number defined by the bits \(b\) will be \(\sum b_i 2^i\), and a similar convention will be adopted for byte lists.
Initialization

We start with some auxiliary conversion functions that will be used by the Keccak algorithm but are not intended to be called directly by the user. Thus `bytestohexstring` converts a list of bytes (as integers in the 0..255 range) to a hex string and `hexstringtobytes` reverses this process. Similarly, `filetobytes` reads a binary file to a list of bytes and `bytestofile` writes a list of bytes to a file. Some of these functions are combined in `messagetobytes`, which on input a message given either as an ASCII text string, a hex string or a file, converts it to a list of bytes suitable to be processed by the SHA-3 functions. For speed, these functions use tables to convert between bytes given as integers and the corresponding hexadecimal strings.

```maple
restart:

bytestohex := proc(i) if i < 16 then cat(0", StringTools:-LowerCase(convert(i, hex))) else StringTools:-LowerCase(convert(convert(i, hex), string)) end if end proc:

bytestohexstring := proc(bytelist :: (list(integer[0 .. 255]), integer[0 .. 255])) if bytelist = [ ] then "" else cat(op(map(x -> bytestohex[x], bytelist))) end if end proc:

hextobytes := table(map(x -> x = convert(x, decimal, hex), convert(bytestohex, list))):

hexstringtobytes := proc(str :: string) if str = "" then [ ] else map(x -> hextobytes[x], [StringTools:-LengthSplit(str, 2)]) end if end proc:

filetobytes := proc(filename :: string) local f;
  f := FileTools:-Binary:-Read(filename, integer[1]) mod 256;
  FileTools:-Binary:-Close(filename);
  f
end proc:

checkfile := proc(filename :: string)
The next function, checkkey, converts a key, given either as a text or a hex string, to a list of bytes that can be used by the HMAC function. If the string contains some non-hex character, then it is automatically regarded as an ASCII text string, otherwise it is regarded as a hex string. We have

```plaintext
local r, f;
f := filename;
if FileTools:-Exists(f) then
    r := readstat("File exists, o/c/a to overwrite/change/abort");
    if r = o then
        error "Operation aborted by user"
    else
        f := readstat("Please enter new file name:");
        f := procname(f)
    end if
end if;
f;
end proc:
```

```
bytetointeger1table := Array(0 .. 255, x → mods(x + 1, 256) − 1):

bytestorefile := proc(l :: list(integer[0 .. 255]), filename :: string, filecheck :: truefalse := true, logoutput :: name := terminal)
local f, n;
f := filename;
if filecheck then f := checkfile(f) end if;
    # Check if a file named filename already exists in the current directory
    n := nops(l);
    FileTools:-Binary:-Write(l, filename, x → bytetointeger1table[x], l);
    FileTools:-Binary:-Close(f);
if logoutput = file then
    writeto(cat( filename, ".log" ));
    printf("%d bytes saved to %s\n\n", n, f);
    writeto(terminal)
elsif logoutput = terminal then
    printf("%d bytes saved to %s\n\n", n, f)
else
    NULL
end if
end proc:
```

```
messagebytess := proc(message :: string, messagetype :: name)
if messagetype = file then
    filetobytes(message)
elsif messagetype = text then
    convert(message, bytes)
elsif messagetype = hex then
    hexstringtobytes(StringTools:-LowerCase(message))
else error "incorrect message type specification"
end if;
end proc:
```

The next function, checkkey, converts a key, given either as a text or a hex string, to a list of bytes that can be used by the HMAC function. If the string contains some non-hex character, then it is automatically regarded as an ASCII text string, otherwise it is regarded as a hex string. We have
imposed a lower minimum of 128 bits for the key, so that the supplied key will be rejected if the hex string has less than 32 characters or if the text string has less than 16 characters. The maximum length will be 512 bits and if the supplied string is longer it will be truncated to this length (i.e., to 128 characters if it is a hex string and to 64 characters otherwise). Of course, all these conventions and restrictions are easy to modify and, in particular, it is possible to define a key type as hex or text according to whether we want the string to be regarded as a hex or a text string and the size limits can also be easily modified or even lifted. However, we feel that these restrictions are reasonable and make the functions easier to use. The function \textit{checkkey} takes as input a key given as a string (either hex or ASCII) and, after checking that the string length is no less than 128 bits, converts it to a list of bytes (at most 64 bytes, corresponding to a 512-bit key as mentioned above); if the supplied key is too short, then it is rejected as invalid.

\begin{verbatim}
checkkey := proc(key :: string)
  uses StringTools;
  local k;
  k := key;
  if not IsHexDigit(k) then
    if Length(k) < 16 then
      error "The supplied key has less than 128 bits"
    elif Length(k) > 64 then
      k := Take(k, 64)
    end if;
  end if;
  k := convert(k, bytes)
else
  if Length(k) < 32 then
    error "The supplied hex key has less than 128 bits"
  elif Length(k) > 128 then
    k := Take(key, 128)
  elif Length(k) mod 2 = 1 then
    k := cat("0", k)
  end if;
  k := hexstringtobytes(LowerCase(k))
end if;

k
end proc:
\end{verbatim}

The next function performs a cyclic bitwise shift on an integer $n$. The parameter $l$ sets the integer length, i.e., the number of bits that are going to be shifted (this is necessary because an integer that is represented, say, by a list of 64 bits, may have one or several leading bits equal to 0 and the result of applying the cyclic rotation would be different if these 0 bits were not taken into account). Finally, the parameter $s$ (where we may assume that $s$ is less than $l$ or, otherwise, reduce $s$ modulo $l$) sets the number of positions that the bits are going to be shifted.

\begin{verbatim}
Rot := proc(n, l, s)
  local t;
  irem(n, 2^{l-s}) \cdot 2^{s} + iquo(n, 2^{l-s})
end proc:
\end{verbatim}

We will use the following function to convert a lane, given as a list of bytes, to a $w$-bit integer, note that for this conversion we use the little-endian convention.
Finally, a function to convert a decimal integer $x$ to base $b$, where the base-$b$ representation is the list of digits in little-endian order, and $l$ is a parameter that specifies the length of this list, so that if the integer has less than $l$ digits in base $b$, the list of digits given by the function is completed with zeros on the right up to the required length (we will always assume that this length is greater than or equal to the bit-length of $x$).

```plaintext
> byte2oint := proc(bytes)
local l;
    l := nops(bytes);
    add(bytes[i]*256^i-1, i = 1 .. l)
end proc:
```

The Keccak sponge function

We start the implementation of the Keccak sponge function by giving the Round Constants, which are used in the $i$ (iota) step of each round of Keccak-$f[b]$. The 24 constants are initialized as hex strings in a one-dimensional array:

```plaintext
> RChex := Array(0 .. 23, ["00000000000000001",
    "0000000000000008082",
    "80000000000000808a",
    "80000000080008000",
    "00000000000000808b",
    "000000000000000001",
    "80000000080008001",
    "800000000000000809",
    "000000000000000008a",
    "0000000000000000088",
    "0000000000000000089",
    "0000000000000000080a",
    "0000000000000000080b",
    "80000000000000000b",
    "800000000000000009",
    "800000000000000003",
    "800000000000000002",
    "800000000000000000",
    "000000000000000000a",
    "800000000000000000803",
    "8000000000000000802",
    "8000000000000000800",
    "000000000000000000a",
    "80000000000000000080a",
    "80000000000000000080b",
    "800000000000000000809",
end proc:
```
We now convert each round constant to an integer that will be XORed with the first lane of the state in the \(t\) step. For this, according to the little endian convention we have mentioned, the bytes in each constant are reversed before converting them to an integer. Later, in the absorption phase, these integers will be automatically truncated to lane size \(w\) whenever \(w < 64\), which happens when \(b < 1600\) in order to XOR them with other lanes; this truncation will be forced by the use of \texttt{Bits:-Settings(dafaultbits = w)} in the function \texttt{keccak} below.

\[ RCint := \text{bytestoint}\left(\text{ListTools:-Reverse}\left(\text{hexstringtobytes}\left(\text{RChex}\right)\right)\right); \]

Next, we set up the cyclic shift offsets that are used in the \(\rho\) and \(\pi\) steps of each round of \texttt{Keccak-f[b]}. The offsets are the elements of the following two-dimensional array:

\[ ro := \text{Array}(0..4, 0..4, [[0, 36, 3, 41, 18], [1, 44, 10, 45, 2], [62, 6, 43, 15, 61], [28, 55, 25, 21, 56], [27, 20, 39, 8, 14]]); \]

The next step is to build the round function, for which we shall assume that the lanes are given as integers. The bitwise operations on these integers required by the \texttt{Keccak} permutation are then performed using the Maple package \texttt{Bits} (plus the function \texttt{Rot} we defined before to perform the cyclic shift) and it is important to notice that, for these operations to work properly, it is necessary to set the parameter \texttt{defaultbits} of the \texttt{Bits} package to the lane length \(w\) which, as mentioned, is done in the function \texttt{keccak} below. Note that Maple's convention is to number the list elements starting from 1 (instead of the numbering starting from 0 used in the \texttt{Keccak} reference) but this does not cause any trouble. The bitwise cyclic shifts used in the \(\rho\) and \(\pi\) steps are right shifts according to the little endian convention (bits are shifted from less to more significant) but sometimes they are called "left shifts" because the big endian convention is assumed. The cyclic right shift of size \(s\) is obtained with \texttt{Rot}. We remark that the Maple code for this function is an almost word-for-word rendition of the pseudocode given in \texttt{Kimplementation}.

The inputs of the \texttt{Round} function are the state \(A\), the permutation width \(b\), and the round number \(rn\). The output is the modified state.

\[ \textbf{Round} := \text{proc} \left( A :: \text{Array}, b :: \text{posint}, rn :: \text{nonnegint} \right) \]
\[ \text{uses} \ \texttt{Bits}; \]
\[ \text{local} \ w, wb, B, C, D, x, y; \]
\[ w := \text{iquo}(b, 25); \]
\[ B := \text{Array}(0..4, 0..4, \text{fill = 0}); \# \text{the zero (or root) state} \]
\[ C := \text{Array}(0..4, \text{fill = 0}); \]
\[ D := \text{Array}(0..4, \text{fill = 0}); \]
\[# \text{theta step:} \]
\[ \text{for} \ x \ \text{from} \ 0 \ \text{to} \ 4 \ \text{do} \]
\[ \ C[x] := \text{Xor}(\text{Xor}(\text{Xor}(A[x, 0], A[x, 1], A[x, 2]), A[x, 3]), A[x, 4]) \]
\[ \# \text{The following is a slightly slower alternative:} \]
\[ \ C[x] := \text{foldl} \left( \text{Xor}, A[x, 0], A[x, 1], A[x, 2], A[x, 3], A[x, 4] \right) \]
\[ \text{end do}; \]
The next step is to build the Keccak-f\[b\] permutation, which consists simply in applying the successive round functions to the state. The inputs are a state \(A\) and the permutation width \(b\); the output is the modified state.

Keccak uses multi-rate padding, an algorithm that pads the message given as a bit list by appending a single bit 1 followed by the minimum number of bits 0 followed by a single bit 1 such that the resulting length is a multiple of the bit rate \(r\) (see Kreference). The next function implements this algorithm but adds two variants: for the SHA-3 hash functions the application of multi-rate padding is preceded by appending first the bits "01" to the message bit list and for the SHA-3 XOF's, the four bits "1111" are appended first. Having the full message given as a list of bits is not convenient if the message is long and hence we will consider the message as a list of bytes (which is not restrictive in this context, given the byte-oriented approach mentioned before). These blocks will
later be divided into lanes and these lanes converted to integers in the Keccak function below, just before their absorption into the state.

The inputs of the padding function are the message given as a list of bytes (i.e., a list of integers in the 0..255 range), the bit rate \( r \) and \( \text{domain} \), which can take the values 'hash', 'xof' or 'kec' to apply different paddings and implement domain separation by distinguishing the inputs corresponding to the SHA-3 hash functions from these corresponding to the SHA-3 XOFs, with the value 'kec' corresponding to the Keccak function itself. The output is an array containing the padded message blocks, each of which is given as a list of bytes.

```plaintext
> pad := proc(message :: list, r :: posint, domain :: identical(hash, xof, kec))
  local rby, numby, aby, blocks, lastblock, last;
  rby := iquo(r, 8);
    #This is the bit rate in bytes (byte rate)
  numby := nops(message);
    #The number of bytes in the message
  aby := irem(numby, rby);
    #Either 0 or the number of remaining bytes after partitioning in blocks
  blocks := [ListTools:-LengthSplit(message, rby)];
    #The partitioned message, now padding starts ...
  blocks := Array(1..nops(blocks), i->blocks[i]);
  if aby = 0 then
    if domain = hash then
      lastblock := [6, 0$(rby - 2), 128]
    elif domain = xof then
      lastblock := [31, 0$(rby - 2), 128]
    else
      lastblock := [1, 0$(rby - 2), 128]
    end if;
  if numby = 0 then
    blocks := Array(1..1, fill = lastblock)
  else
    blocks := ArrayTools:-Concatenate(2, blocks, Array(1..1, fill = lastblock));
  end if;
  elif aby = rby - 1 then
    if domain = hash then
      blocks[-1] := [op(blocks][-1]), 134]
    elif domain = xof then
      blocks[-1] := [op(blocks][-1]), 159]
    else
      blocks[-1] := [op(blocks][-1]), 129]
    end if;
  else
    #In this case at least two bytes are added to the last block
    if domain = hash then
      blocks[-1] := [op(blocks][-1]), 6, 0$(rby - aby - 2), 128]
    elif domain = xof then
      blocks[-1] := [op(blocks][-1]), 31, 0$(rby - aby - 2), 128]
    else
      blocks[-1] := [op(blocks][-1]), 1, 0$(rby - aby - 2), 128];
    end if;
  end if;
end proc:
```
Next we give the function that absorbs one block into the state. For this we use Maple's function `ArrayTools:-Copy`, which copies a one-dimensional array containing a Keccak block into the zero state. The entries of the block are copied to the state in Fortran order, which Maple uses by default, and this corresponds exactly to the mapping defined in [Kreference], so that we don't need to make the mapping function explicit (note, however, that if we represent the 5x5 state array as a matrix, the $x$-coordinate serves to enumerate rows and the $y$-coordinate to enumerate columns, in contrast with the graphical picture given in [Kreference, p. 11], where the $x$-coordinates belong to the horizontal axis and the $y$-coordinates to the vertical one. Independently of this pictorial representation, the indices of the block are mapped to those of the state in the following order: $(0, 0), (1,0), (2,0), (3,0), (4,0), (0,1), (2,1), ...$, which is precisely Fortran order. The function inputs are the state array, the block given as a list of bytes, the bit rate $r$ and the permutation width $b$. The output is the modified state.

```maple
AbsorbBlock := proc(S, blocklist, b, r)
local B, w, T;
B := S;
w := iquo(b, 200);
T := Array(0..4, 0..4, fill = 0);
ArrayTools:-Copy(Array(bytestoint~([ListTools:-LengthSplit(blocklist, w)])), T);
B := Bits:-Xor~(B, T);
B := Keccakf(B, b)
end proc;
```

The following is a different version of the absorbing function which is functionally equivalent to the previous one but does not employ the resources of Maple's `ArrayTools` package and is closer in spirit to the description in Kimplementation. To force the function `keccak` below to use it instead of the previous function it suffices to drop the 2 at the end of the name and to rename it to `AbsorbBlock`.

```maple
AbsorbBlock2 := proc(S, blocklist, b, r)
local B, w, z, m, n, x, y;
B := S;
w := iquo(b, 200);
z := iquo(iquo(r, 8), w);
m := iquo(z, 5);
n := irem(z, 5);
for x from 0 to 4 do
  for y from 0 to m - 1 do
    B[x,y] := Bits:-Xor(B[x,y], bytestoint(blocklist[w .. (5*y + x + 1) + 1 .. w .. (5*y + x + 1)]));
  end do;
end do;
for x from 0 to n - 1 do
  B[x,m] := Bits:-Xor(B[x,m], bytestoint(blocklist[w .. (5*m + x + 1) + 1 .. w .. (5*m + x + 1)]));
end do;
B := Keccakf(B, b)
end proc;
```
Next we give the squeezing function, which extracts the bits from the state after processing them with the Keccak permutation. The inputs are the state array, the Keccak width, the bit rate and the required output length in bits. The output is a list of bytes corresponding to the required length.

\[ \text{Squeeze} := \text{proc}(S, b, r, n) \]
\[ \text{uses ListTools;} \]
\[ \text{local } w, z, B, l, m, Z, h; \]
\[ w := \text{iquo}(b, 25); \]
\[ B := S; \]
\[ l := \text{convert}(B, \text{list})[1..\text{iquo}(r, w)]; \]
\[ \text{while } \text{nops}(l) < \frac{n}{w} \text{ do} \]
\[ B := \text{Keccakf}(B, b); \]
\[ l := [\text{ap}(l), \text{ap}(\text{convert}(B, \text{list}))[1..\text{iquo}(r, w)]] \]
\[ \text{end do;} \]
\[ l := \text{Flatten}(\text{decimalto}~(l, 256, \text{iquo}(w, 8))); \]
\[ l[1..\text{iquo}(n, 8)] \]
\[ \text{end proc;} \]

Now we combine the previous functions to obtain a version of Keccak that acts on messages given as lists of bytes and produces its output also in this format. This function is used below to construct the Keccak sponge function and the SHA-3 functions, as well as the SHA-3 based HMAC. As mentioned above, all of the numeric parameters \( b, r, n \), are required to be multiples of 8 as a consequence of the byte-oriented character of the implementation. In fact, we will require that the bit rate \( r \) is a multiple of the lane width \( w \) since this simplifies the implementation somewhat and is not too restrictive in practice (for example, all the values for \( b, r \) and \( n \) used in the SHA-3 functions, and hence on HMAC based on these functions, comply with this requisite). Here the command \text{Bits: } \text{Settings}(\text{defaultbits} = w) \text{ plays an important role because it forces all bitwise operations with the integers representing } w\text{-bit lanes to be performed with } w\text{-bit width and, in particular, if the lanes are shorter than 64 bytes (as it happens whenever } b < 1600), \text{ the XOR operation with the 64-bit round constants defined above is performed over the first } w\text{ bits, which is equivalent to the truncation of these constants before XORing them and makes it unnecessary to perform this truncation explicitly.} \]

The function proceeds as in the pseudocode given in [Kimplementation, p. 9]. First, padding is performed with the previous function \text{pad} and then the state is initialized. The next step is the absorbing phase which is performed for each block by the function \text{AbsorbBlock} and, finally, the squeezing phase produces the requested output by means of the function \text{Squeeze}. \]

The inputs of the function are the message in one of the three valid formats, the permutation width \( b \), the bit rate \( r \), the output length \( n \) and, finally, \text{domain}, which is used to select the type of padding that will be applied and will take the values 'hash' for the SHA-3 hash functions, 'xof' for the SHAKE functions and 'kec' for Keccak. The output is a list of bytes which will be converted in the function Keccak below to a hex string to obtain the message digest.

\[ \text{keccak := proc(message, b, r, n, domain)} \]
\[ \text{local } m, w, S, nb, i, P; \]
\[ \text{if not } b \text{ in [200, 400, 800, 1600] then} \]
\[ \text{error "b-value of %1 not supported", } b \]
\[ \text{end if;} \]
\[ \text{if not } n \mod 8 = 0 \text{ then} \]
\[ \text{error "the output length } n = \%1 \text{ should be a multiple of 8", } n \]
Finally, we give the Keccak (sponge) function for ordinary messages. The input parameters are message, for a message given either as a text string, a file name (including the full path if the file is not in the current working directory) or a hex string, b for the Keccak width, r for the bit rate, n for the desired output length in bits (all of them satisfying the requirements stated above) and, finally, a name (hex, text or file) to indicate which kind of message is being processed (with text being the default). The default values for the width and the bit rate are those suggested in Ksubmission. As mentioned in Csponge the resulting capacity value of 576 precludes generic attacks with expected complexity below $2^{288}$. The output is the message digest given as a hex string.

```
> Keccak := proc(message :: string, b :: posint := 1600, r :: posint := 1024, n :: posint
   := 1024, messagetype :: name := text)
   uses ListTools;
   local m, l;
   m := messagetobytes(message, messagetype);
   l := keccak(m, b, r, n, kec);
   bytestohexstring(l)
   end proc:
```

### SHA-3

The process of SHA-3 standardization by NIST was completed in August 2015 and includes four hash functions with output lengths of 224, 256, 384 and 512 bits. In all these cases, the width is 1600 (i.e., the underlying permutation is Keccak-f[1600]) and the capacity is twice the output length. The capacity works as a security parameter, so that security is increased with higher capacities, but there is a security-efficiency trade-off and speed may be increased by using lower capacities (see KParameters).

The function SHA3 below is an implementation of the four hash functions in the SHA-3 standard. It is an indexed function which should be called in the form:

```
SHA3[n](message)
```

where $n$ is the output length in bits (which must be one of the following values: 224, 256, 384, 512),
message is the message and there is an additional (optional) input parameter to specify the type of message (text, hex or file) with text the default.

> SHA3 := proc(message :: string, messagetype :: name := text)
>     local n, m, l;
>     if type(procname, 'indexed') then
>         n := op(procname)
>     else
>         error "output length not specified"
>     end if;
>     if not n in {224, 256, 384, 512} then
>         error "%1 is not a valid output length", n
>     end if;
>     m := messagetobytes(message, messagetype);
>     l := keccak(m, 1600, 1600 − 2·n, n, hash);
>     bytestohexstring(l)
> end proc:

The next procedure implements the XOFs SHAKE128 and SHAKE256. These functions should be called in the form:

SHAKE[n](message, d)

where n is now the "security strength" of the function which must be either 128 or 256, while d represents the bit-length of the output, which should be a multiple of 8.

> SHAKE := proc(message :: string, d :: integer, messagetype :: name := text)
>     local n, m, l;
>     if type(procname, 'indexed') then
>         n := op(procname)
>     else
>         error "output length not specified"
>     end if;
>     if not n in {128, 256} then
>         error "%1 is not a valid security strength parameter", n
>     end if;
>     m := messagetobytes(message, messagetype);
>     l := keccak(m, 1600, 1600 − 2·n, d, xof);
>     bytestohexstring(l)
> end proc:

HMAC-SHA-3

One of the most important applications of hash functions is to the construction of MACs (Message Authentication Codes). MACs are cryptographic schemes designed to prevent an adversary from impersonating a legitimate user and from modifying a message without the legitimate users noticing it. For this purpose, two users share a secret key k and when one of them
wants to send a message $m$ to the other, she uses a tag generation algorithm that computes a MAC tag $t$ using $m$ and $k$ as inputs and sends the pair $(m,t)$ to the receiver. The receiver verifies that $t$ is a valid tag for $m$ by running a verification algorithm that, on input $m$, $k$ and $t$, checks whether $t$ is a valid tag for $m$ (this is done by computing the tag from $m$ and $k$ and checking whether it is equal to the received tag). If the message has been modified by an adversary, this will be noticed by the receiver because the adversary, not knowing the secret key, will be unable to compute a valid tag for the message if the MAC is secure (see, for example, [Gómez Pardo] for the precise definition of security for MACs). Thus a secure MAC gives the receiver assurance that the message has not been modified (thus providing message integrity) and also that the message is authentic (for only the legitimate user in possession of $k$ should be able to generate a valid tag for $m$).

One of the basic ideas to obtain a MAC from a cryptographic hash function consists in computing the message tag by hashing the key together with the messages. There are several concrete ways of doing this and the most popular one gives rise to HMAC (Hash-based MAC, introduced in [Bellare-Canetti-Krawczyk]) which, using a hash function $H$, combines the message with the key and some padding and computes the tag $t = HMAC(k,m) = H((k\oplus\text{opad})||H((k\oplus\text{ipad})||m)))$, where $\oplus$ denotes the bitwise XOR, $\|\$ denotes concatenation of strings, $\text{ipad}$ is a string of bytes 0x36, and $\text{opad}$ a string of bytes 0xc6, both of length equal to the block size $B$, a parameter associated to the scheme (we omit here some details and refer to the HMAC papers page, or to [Gómez Pardo, Algorithm 5.4], or also to the Wikipedia HMAC page for a more detailed description). HMAC is used in both the Transport Layer Security (TLS) protocol used by web browsers for secure connections, as well as in the IPSec security suite. It is regarded as a secure scheme although its practical security in some settings has been questioned in [Koblitz-Menezes].

Although HMAC-SHA-3 has not been standardized and the Keccak authors mention in their web page that "Unlike SHA-1 and SHA-2, Keccak does not have the length-extension weakness, hence does not need the HMAC nested construction. Instead, MAC computation can be performed by simply prepending the message with the key", we give here a SHA-3-based implementation of HMAC (see, for example, SHA3-based MACs) which uses any one of the four SHA-3 hash functions defined above. The parameters $B$ (input block length in bytes) and $L$ (output size in bytes) are computed in the following table (see FIPS-198-1 for a detailed description of the NIST HMAC standard or also [Gómez Pardo, Section 5.6.5]). In this case, following the recommendation in [Ksubmission, 5.1.3, p. 9], $B$ is equal to the byte rate and $L$ the output length of the hash function in bytes, which is one of the following values: 28, 32, 48 or 64. The table is indexed by the four output bit lengths and the entry corresponding to each of these lengths is an array containing the values of $B$ and $L$ (in this order) associated to it.

> \[ \text{HIT} := \text{table} \left( \text{map} \left( x \rightarrow x = \text{Array} \left( 1..2, \left[ 200 - \frac{x}{4}, \frac{x}{8} \right] \right), \left[ 224, 256, 384, 512 \right] \right) \right); \]

The next function, HMAC3, computes the SHA-3-based HMAC tag for a message. Like the function SHA3, HMAC3 is an indexed function which is called in the form:

HMAC3[$n$(key, message)

where $n$ is the output length in bytes and must be equal to one of the following: 224, 256, 384, 512. The key, given either as a text or a hex string, is processed with the function checkkey above and hence it must have a minimum bit length of 128 (this seems a basic security requirement and, in fact, in HMAC it is advisable to use a key length greater than or equal to the output length). However, if desired it is easy to modify the function to allow for the use of smaller keys (for example, for testing purposes). There is no maximum length for the key but keys longer than 512
bits will be truncated to this length by checkkey (a key length greater than 512 would not significantly improve security and would, on the other hand, have obvious practical disadvantages since keys have to be shared through secure channels and stored safely). The message can be an ASCII string, a hex string or a file, with the type specified through the optional parameter messagetype, which has text as default.

> \( \text{HMAC3} := \text{proc}(key :: \text{string}, message :: \text{string}, \text{messagetype} :: \text{name} := \text{text}) \)
  
  _local n, B, L, r, m, k, ipad, opad;
  _if type(procname, 'indexed') then
      _n := op(procname)
  _else
    _error "output length not specified"
  _end if;
  _if not n in \{224, 256, 384, 512\} then
    _error "%1 is not a valid output length", n
  _end if;
  _B := \text{HT}[n][1];
  _L := \text{HT}[n][2];
  _r := 8 \cdot B;
  _m := \text{messagetobytes}(message, \text{messagetype});
  _k := \text{checkkey}(key);
  _k := [\text{op}(k), 0S(B - nops(k))];
  _ipad := [54$B];
  _opad := [92$B];
  _m := \text{keccak}([\text{op}(\text{Bits:Xor}-(k, \text{ipad})), \text{op}(m)], 1600, r, n, \text{hash});
  _bytestohexstring(\text{keccak}([\text{op}(\text{Bits:Xor}-(k, \text{opad})), \text{op}(m)], 1600, r, n, \text{hash}))
_end proc:

### KAT tests and other tests

Here we include some Known Answer Tests taken from [Keccak Code Package Test Vectors](#) and [KATests](#). Note, however, that the tests in the latter reference have not been updated and hence give different results for SHA-3 and SHAKE because they do not take into account the new padding method introduced by NIST. We start with a few short hex messages.

> \text{sm0} := "";
  
  \text{sm16} := "41FB";
  
  \text{sm64} := "4A4F202484512526";
  
  \text{sm320} :=
    "7ABAA12EC2A7347674E444140AE0FB659D08E1C66DECD8D6EA925FA451D65F\3C0308E29446B8ED3"
  ;
  
  \text{sm800} :=
    "433C5303131624C0021D868A30825475E8D0B0D3052A022180398F4CA4423B98214B\6BEAAC21C8807A2C33F8C93BD42B092CC1B06CEDF3224D5ED1E2C2978444F22F2\08A55A5A58542B524B02CD3D55D5F6907AFE71C5D7462224A3F9D9E53E7E0846D\BB4CE" ;
Here, we use the Keccak function with $b = 200$, $r = 40$, and $c = n = 160$. In this case the output length is higher than the bit rate, so that the squeezing phase requires more than one application of the Keccak permutation:

```plaintext
> Keccak~([sm0, sm16, sm64, sm320, sm800, sm2040], 200, 40, 160, hex);
```

Next, with different parameters:

```plaintext
> Keccak~([sm0, sm16, sm64, sm320, sm800, sm2040], 800, 512, 512, hex);
```

Let us now compute the hashes of the preceding messages with the four SHA-3 functions:

```plaintext
> SHA3[224]~([sm0, sm16, sm64, sm320, sm800, sm2040], hex);
```

```plaintext
> SHA3[256]~([sm0, sm16, sm64, sm320, sm800, sm2040], hex);
```

```plaintext
> SHA3[384]~([sm0, sm16, sm64, sm320, sm800, sm2040], hex);
```

```plaintext
> SHA3[512]~([sm0, sm16, sm64, sm320, sm800, sm2040], hex);
```

Let us now apply the SHAKE functions to some of these messages setting the output length equal to 4096 bits; the results agree with those in Keccak Code Package Test Vectors.

```plaintext
> SHAKE[128]~([sm0, sm64, sm320, sm2040], 4096, hex);
```

```plaintext
> SHAKE[256]~([sm0, sm64, sm320, sm2040], 4096, hex);
```
The following messages from Keccak Code Package Test Vectors have byte lengths equal to $B-1$, $B$, and $B+1$, where $B = 136$ is the byte length of the blocks in which the message is partitioned when a bit rate $r = 1088$ is used. Thus they serve to test the different cases in the pad function according to the number of remaining bytes after partitioning the message in blocks of this length.

```plaintext
> sm1080
 := "B771D5CEF5D1A41A93D15643D7181D2A2E0F0A8E84D91812F20ED21F147BEF732\BF3A6EF4067C3734B85BC8CD47180F10D9E8291B5839A677B960218F71E793\F2797AEA394906512829065D37BB55EA796FA4F56FD8896B49B2CD19B43215AD9\67C712B2E5032D65232E02C127409D2ED416B975D763D52DB98D949D3B0FEB\D6A8052FBB":

sm1088
 := "B32D95B0B9AAD2A8816DE6D06DF86008505DB8C14124F6E9A163B5A2ADE55F835DFD388ECF50F700D3B25E42CC0AF050C4D1BE5E555B23087E04D7BF9813\622780C7313A1954F8740B6EE2D3F71F768D417F520482DB3A08DF4F22B4EE9DB\D015447B35307DD50F3A4247C5DE9A8AB6D2A8DECEA01E3B87C8B9275B08B\EB37674C6F8EE380C4":

sm1096
 := "04410E31082A47584B406F051938A6AABE74E4DA59BB685E6B49E8A1F7F2CA00DF\FBA5462C2CD2BFDE8B64FB21D70C083F1118B56A52D3B81AC5EEC29E3B31B\D0078B6156786DA36D68C3098B5C47BB67AC64DB14165A6F5B4A4544D8066DEF\F487D5373C7F9792C299E9686B7E5821E7C8E2458315B996B5677D926DAC57B3F2\2DA873C601016A0D":
```

The result of applying SHAKE256 (which uses Keccak with bit rate $r = 1088$) to these messages is the following (the same as in Keccak Code Package Test Vectors):

```plaintext
> SHAKE[256] ~ ([sm1080, sm1088, sm1096], 4096, hex);

Next we give a "long message", taken from KATests.

```plaintext
> lm34304
 := "0EBF64AC017FEBDCA40FF85FD4AE88FA827561C150F74CD5E864857FBB\9C08A46EBB9FF7A16919618C9FB06BF8E3F0859774DA6C38C5A0C54DA4D075D1B\A6F1482B7705C8E1A86E790BF0328246B5E6BE013F934DECC34808A3639C4946\4309DF5A250BF4521E41B4CDE56356BB8625076AD7E26018EEF2325D15F36B\DBBCEC374758C68A0E72D83A37DB4D2022A0AF6B16F4515053E1F9398CC6A7\4D3434C4B03597AE6833E1FE8F93F756F6554F98CBCC84702D19F875D347C\345B09EDFCB1C71BD6955C1578DFC07376728FCC3AB9565C0A1A8DCA7822102\8B00B51B175A2DA2C0A90C33C169EA8A1A2E375C087AC3657D28AC841D5B5C\225AEEFA85919FBA628F3F42F1FC0806A250143C084322FE9E30BCC8B89F3DA734A69CEB935EE2587D84321CC8A8BB3C7515F4894CBB573178232A5631979BBEB\AEACDFBF711A84F83BB89E0F922B88C5EE83E4F93AC34CF80862A17ED5549B77ED7\939C368CD9A92EAAE595105DE7E2889CADD71D01201580038FCFCA4B4D258A13BE\FAAA14FACF5099A83F9E71D515219D055F44F757757BCBDD2734CB91C5C4E7\B411D1E83B42689FDF8A69E63D6DFF255B977D7435BEE5D5EBBA7A47B19CD\AB3F1C40E79B926A481A1629A818525C2A198983F23F0DA5DA8B99633689292332\`
```
The 256-bit and 512-bit hashes corresponding to this message under the SHA-3 functions are the following:

\[ SHA3[256](lm34304, \text{hex}); \]

\[ SHA3[512](lm34304, \text{hex}); \]

We next apply Keccak, with \( b = 1600, r = 576 \) and output length 512:

\[ Keccak(lm34304, 1600, 576, 512, \text{hex}) \]

Next, we include a couple of text messages and we compute their message digests with the four SHA-3 functions; these results can be checked against those obtained using the online implementation of SHA-3 at \texttt{http://emn178.github.io/online-tools/}. Observe that a small change in the message produces a random-looking change in the hash value.

\[ m1 := "The quick brown fox jumps over the lazy dog" ; \]

\[ m2 := "The quick brown fox jumps over the lazy dog." ; \]

\[ SHA3[224] \sim ([m1, m2]); \]

\[ SHA3[256] \sim ([m1, m2]); \]

\[ SHA3[384] \sim ([m1, m2]); \]
We now apply the SHAKE functions to both text messages:

> SHAKE[128]~([mtl, mt2], 256)

> SHAKE[256]~([mtl, mt2], 512)

We create, in the current directory, a file containing the sentence "The quick brown fox jumps over the lazy dog":

> bytestofile(convert(mlt, bytes), "sha3testfile", false);

We compute the hash of the file with SHA3-512; the result is the same that the one obtained for the text string:

> SHA3[512]("sha3testfile", file);

As another example, we compute the SHA3-512 hash of the original file containing this worksheet, namely, "SHA-3.mw", assuming it located on the current directory:

> SHA3[512]("SHA-3.mw", file);

An example of the use of HMAC3 for authentication would be the following. Suppose that Alice wants to send Bob the following message:

> msg := "The meeting will be tomorrow at 11:30 AM":

Alice wants to make sure that the message that Bob receives is exactly this one and that Bob knows that the message really comes from Alice. For authentication purposes they have agreed, through a secure channel, to use the 256-bit key with HMAC3[512]:

> k := "e73a4875sff3dcca50cb049ceceeb3a5e230d45a640711dcdbdd012576ffe6f20":

Alice then computes:

> tg := HMAC3[512](k, msg);

Alice then sends Bob the pair (msg, tg). By repeating the previous computation using the secret key k they share, Bob observes that the value obtained coincides with tg, which gives him assurance
that the message really comes from Alice and has not been modified en route. In this case the message was sent in the clear but it is also possible to combine authentication with an encryption scheme to obtain authenticated encryption.

Next, we use HMAC3 to compute a couple of tags. First we compute the HMAC tag for an ASCII message using SHA3-256:

\[
\text{HMAC3}[256]("0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b0b", "Hi There")
\]

Now, the tag of a hex message using HMAC-SHA3-512:

\[
\text{HMAC3}[512]("aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa", \\
"dddddddddddddddddddddddddddddddddddddddddd", \text{hex})
\]

Another example is the following:

\[
\text{HMAC3}[512]("0102030405060708090a0b0c0d0e0f10111213141516171819", \text{StringTools}:-\text{Repeat}("cd", 50), \text{hex})
\]

Finally, we compute the HMAC-SHA3-512 tag of the file "sha3testfile" created above, using the sentence "This is my not so secret key" as key:

\[
\text{HMAC3}[512]("This is my not so secret key", "sha3testfile", \text{file})
\]

\section*{Conclusions}

We have used Maple's capabilities as a programming environment, to implement and explore the cryptographic hash functions and the extended-output functions standardized by NIST in SHA-3 Standard. In addition, we have also implemented HMAC-SHA-3, the version of HMAC (Hash-based Message Authentication Code) based on the SHA-3 hash functions. These implementations use, whenever possible, lookup tables for efficiency and are able to provide hashes and to authenticate messages given either as hex strings, ASCII text strings or files. We have also included some Known Answer Tests to check compliance with the proposed standard.

\section*{References}

M. Bellare, R. Canetti, H. Krawczyk, Keying Hash Functions for Message Authentication, available at \url{http://cseweb.ucsd.edu/~mihir/papers/kmd5.pdf}


G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, Note on Keccak parameters and usage (Kparameters), available at http://keccak.noekeon.org/NoteOnKeccakParametersAndUsage.pdf


Legal Notice: The copyright for this application is owned by the authors. Neither Maplesoft nor the authors are responsible for any errors contained within and are not liable for any damages resulting from the use of this material. This application is intended for non-commercial, non-profit use only. Contact the authors for permission if you wish to use this application in for-profit activities.