# Exponentiation modulo n

Carsten Urbach

Note that for this we roughly follow the paper by Vedral, Barenco and Ekert (1996). However, we use the addition by qft instead (Draper 2000) of the procedure using Toffoli gates.

## Multiplication modulo $2^n$

For multiplying a state $|j\rangle$ with a constant $a$ we can follow the scheme to multiply two numbers in binary representation. As an example, multiply 5 with 3. 5 has binary representation 0101 and 3 has 0011. So, the procedure for $5 \cdot 3$ is

```
  00011 * 1 (1*3)
+ 00110 * 0 (0*6)
+ 01100 * 1 (1*12)
+ 11000 * 0 (0*24)
= 01111 (15)
```

Now, if we have a controlled add operation, we can use the qubits of the first register (in this case representing 5) as control bits and the other register as the constant to add. The single terms in the sum can be efficiently pre-computed classically as follows

```
summands <- function(x, n, N) {
  b <- as.integer(intToBits(x))
  ret <- c()
  for(i in c(1:N)) {
    s <- 0
    for(j in c(1:N)) {
      s <- s+as.integer(b[j])*2^(i+j-2)
    }
    ret[i] <- s %% n
  }
  return(ret)
}
```

Example

```
x <- 3
summands(3, 2^3, 3)
```

```
[1] 3 6 4
```

Now we need a controlled add operation. Here we can build on our add operation using the qft for which we need a controlled phase shift operation first

```
cRtheta <- function(bits, theta=0.) {
  cqgate(bits=bits, gate=methods::new("sqgate", bit=as.integer(bits[2]),
                                      M=array(as.complex(c(1, 0, 0, exp(1i*theta))),
                                              dim=c(2,2)), type="Rt"))
}
```

```
cadd <- function(c, bits, x, y) {
  n <- length(bits)
  z <- cqft(c=c, x=x, bits=bits)
  for(i in c(1:n)) {
    z  <- cRtheta(bits=c(c, bits[i]), theta = 2*pi*y/2^(n-i+1)) * z
  }
  z <- cqft(c=c, x=z, inverse=TRUE, bits=bits)
  return(invisible(z))
}
```

Let's check whether it works as expected (keep in mind that the register is 3 qubit wide plus 1 control qubit, so addition is modulo $2^3 = 8$):

```
basis <- c()
for(i in c(0:(2^4-1))) basis[i+1] <- paste0("|", i %/% 2, ">|", i %% 2, ">")
x <- H(1)*qstate(4, basis=basis)
c <- 1
bits <- c(2:4)
z <- cadd(c=c, bits=bits, x=x, y=5)
z
```

```
   ( 0.7071068 )    * |0>|0>
 + ( 0.7071068 )    * |5>|1>
```

```
z <- cadd(c=c, bits=bits, x=z, y=2)
z
```

```
   ( 0.7071068 )    * |0>|0>
 + ( 0.7071068 )    * |7>|1>
```

```
z <- cadd(c=c, bits=bits, x=z, y=8)
z
```

```
   ( 0.7071068 )    * |0>|0>
 + ( 0.7071068 )    * |7>|1>
```

Equipped with this functionality, we can finally perform a binary multiplication. Note that we need two registers, the first one to store the initial value and the second one to store the final result of the multiplication

```
mult <- function(reg1, reg2, x, y, swap=TRUE) {
  stopifnot(length(reg1) == length(reg2))
  n <- length(reg2)
  s <- summands(y, 2^n, n)
  for(i in c(1:n)) {
    x <- cadd(c=reg1[i], bits=reg2, x=x, y=s[i])
  }
  if(swap) {
    for(i in c(1:n)) {
      x <- SWAP(c(reg1[i], reg2[i])) * x
    }
  }
  return(invisible(x))
}
```

With this we can perform a *reversible* multiplication, which is why we we introduced the SWAP operations at the end. They interchange the two registers. The result is the following

```
basis <- c()
for(i in c(0:(2^3-1))) {
  for(j in c(0:(2^3-1))) {
    basis[i*2^3+j + 1] <- paste0("|", i, ">|", j, ">")
  }
}
x <- X(2)*qstate(6, basis=basis)
x
```

```
   ( 1 )    * |0>|2>
```

```
reg1 <- c(1:3)
reg2 <- c(4:6)
z <- mult(reg1, reg2, x=x, y=3)
z <- X(5) * z
z
```

```
   ( 1 )    * |0>|6>
```

```
z <- mult(reg1, reg2, x=z, y=3)
z
```

```
   ( 1 )    * |6>|2>
```

Let's be a bit more precise here for the multiplication of, say $a$ and $b$ in two registers, both $n$ qubits wide. Starting with both registers in state $|0\rangle$, we can first bring the result register to state $|1\rangle$ using a NOT operation, i.e. $|0\rangle|1\rangle$. Now, we multiply by $a$, which leaves us with state

$$|a \mod 2^n\rangle|1\rangle$$

which we can apply the NOT gate again to reset the result register to state $|0\rangle$ and then we swap the two registers to arrive at

$$|0\rangle|a \mod 2^n\rangle\,.$$

Now we multiply by $b$ getting us to state

$$|a \times b \mod 2^n\rangle|a \mod 2^n\rangle\,.$$

Multiply the result register with the inverse of $a \mod n$ and apply the NOT gate getting us to

$$|0\rangle|a \times b \mod 2^n\rangle\,.$$

The inverse modulo $2^n$ can be computed efficiently in a classical way by the extended Euclidean algorithm

```
eEa <- function(a, b) {
  if(a == 0) return(c(b, 0, 1))
  res <- eEa(b %% a, a)
  return(c(res[1], res[3] - (b %/% a) * res[2], res[2]))
}

moduloinverse <- function(a, n) {
  res <- eEa(a=a, b=n)
  if(res[1] != 1) stop("inverse does not exist!")
  return(res[2] %% n)
}
```

If $a$ and $2^n$ are not coprime, the inverse does not exist. However, for the application we have in mind this is not an issue.

## Working modulo $N$

So far we have assumed that we work modulo $2^n$, where $n$ was dictated by the number of qubits. However, this is not the realistic case. We have to write an adder modulo $N < 2^n$, i.e. $|x\rangle \to |x + y \mod N\rangle$. We can implement this by subtracting $N < 2^n$ whenever needed. We will follow the convention that if $x \geq N$ the operation will be $|x\rangle \to |x\rangle$. Moreover, we will assume $x, y \geq 0$.

To find out, when this subtraction is needed, is a bit tricky. We want to add $y$ to $x$. To decide beforehand, whether or not we have to subtract $N$, we have to check whether $x < N - y$. If not, we have to subtract $N$. If we subtract $N - y$ from this state $|x\rangle$, the most significant qubit indicates whether there occurred an overflow. Using a CNOT gate we can store this info in one ancilla bit $c_1$. Then we add $N - y$ again to retain the original state. Such an operation can be implemented as follows in a controlled manner (control bit $c$, `bits` the bits in state $|x\rangle$ where x is stored, $a$ an ancilla bit and $y$ the value to compare with.

```
cis.less <- function(c, bits, x, c1, a, y) {
  ## add ancilla bit as most significant bit to bits
  b <- c(bits, a)
  n <- length(b)
  ## cadd works modulo 2^n
  z <- cadd(c=c, bits=b, x=x, y=2^n-y)
  ## 'copy' overflow bit
  z <- CNOT(c(a, c1)) * z
  ## add back, resetting ancilla a to |0>
  z <- cadd(c=c, bits=b, x=z, y=y)
  return(z)
}
```

This routine will set the qubit $|c_1\rangle$ to 1 if $|x_{\text{bits}}\rangle$ is smaller than $y$ and leave it at zero otherwise. It uses $|a\rangle$ as ancilla bit and $|c\rangle$ as control bit. Here an example

```
basis <- c()
for(i in c(0:(2^6-1))) {
  basis[i + 1] <-
    paste0("|", i %/% 8 ,">|a=",
    (i %/% 4) %% 2, ">|c1=", (i%/%2) %% 2,
    ">|c=", i%%2, ">")
}

x <- H(1)*qstate(6, basis=basis)
z <- cadd(c=1, bits=c(4,5,6), x=x, y=5)
z
```

```
   ( 0.7071068 )    * |0>|a=0>|c1=0>|c=0>
 + ( 0.7071068 )    * |5>|a=0>|c1=0>|c=1>
```

```
## 5 < 7 -> c1 = 1
v <- cis.less(c=1, bits=c(4,5,6), x=z, c1=2, a=3, y=7)
v
```

```
   ( 0.7071068 )    * |0>|a=0>|c1=0>|c=0>
 + ( 0.7071068 )    * |5>|a=0>|c1=1>|c=1>
```

```
## 5 > 3 -> c1 = 0
w <- cis.less(c=1, bits=c(4,5,6), x=z, c1=2, a=3, y=3)
w
```

```
   ( 0.7071068 )    * |0>|a=0>|c1=0>|c=0>
 + ( 0.7071068 )    * |5>|a=0>|c1=0>|c=1>
```

```
## 5 < 9 -> c1 = 1
w <- cis.less(c=1, bits=c(4,5,6), x=z, c1=2, a=3, y=9)
w
```

```
   ( 0.7071068 )    * |0>|a=0>|c1=0>|c=0>
 + ( 0.7071068 )    * |5>|a=0>|c1=1>|c=1>
```

Now, recall that if $x \geq N$ we want the operation to leave the state unchanged. So, we need two `cis.less` operations, one to check whether $x < N$, which we store in ancilla qubit $c_1$ and another one to check whether $x < N - y$ stored in $c_2$. Note that the combination $c_1 = 0, c_2 = 1$ is not possible. The implementation looks as follows:

```
caddmodN <- function(c, bits, c1, c2, a, x, y, N) {
  stopifnot(length(a) == 1 && length(c1) == 1 &&
            length(c2) == 1 &&
            length(unique(c(c1, c2, a))) == 3)
  y <- y %% N
  ## set c1=1 if x < N
  z <- cis.less(c=c, bits=bits, x=x, c1=c1, a=a, y=N)
  ## set c2=1 if x < N - y
  z <- cis.less(c=c, bits=bits, x=z, c1=c2, a=a, y=N-y)

  ## if c1 and not c2, x = x + y - N
  z <- X(c2) *( CCNOT(c(c1, c2, a)) * (X(c2) * z))
  z <- cadd(c=a, bits=bits, x=z, y=y - N)
  z <- X(c2) * (CCNOT(c(c1, c2, a)) * (X(c2) * z))

  ## if c1 and c2 add x = x + y
  z <- CCNOT(c(c1, c2, a)) * z
  z <- cadd(c=a, bits=bits, x=z, y=y)
  z <- CCNOT(c(c1, c2, a)) * z

  ## reset c1,2
  z <- cis.less(c=c, bits=bits, x=z, c1=c2, a=a, y=y)
  z <- CNOT(c(c1, c2)) * z
  z <- cis.less(c=c, bits=bits, x=z, c1=c1, a=a, y=N)
  return(invisible(z))
}
```

For the reset part: in the first step we flip $c_2$ if $x + y \mod N < y$. This can only be true if $x > N - y$. If $c_2$ was 1, it's zero now and the other way around. The next CNOT gate flips $c_2$ to $|0\rangle$. The last step resets $c_1$ to $|0\rangle$.

You can also see from the routine above that, if $x \geq N$ then `caddmodN` leaves the state unchanged.

Example

```
basis <- c()
for(i in c(0:(2^7-1))) {
  basis[i + 1] <-
    paste0("|", i %/% 16 , ">|a=", (i %/% 8) %% 2,
           ">|c2=", (i %/% 4) %% 2,
           ">|c1=", (i%/%2) %% 2, ">|c=", i%%2, ">")
}

x <- X(1)*qstate(7, basis=basis)
x
```

```
   ( 1 )    * |0>|a=0>|c2=0>|c1=0>|c=1>
```

```
bits <- c(5,6,7)
c <- 1
c1 <- 2
c2 <- 3
a <- 4
N <- 5
z <- caddmodN(c=c, bits=bits, c1=c1, c2=c2, a=a, x=x, y=3, N=N) # 0 + 3 mod 5
z
```

```
   ( 1 )    * |3>|a=0>|c2=0>|c1=0>|c=1>
```

```
z <- caddmodN(c=c, bits=bits, c1=c1, c2=c2, a=a, x=z, y=1, N=N) # 3 + 1 mod 5
z
```

```
   ( 1 )    * |4>|a=0>|c2=0>|c1=0>|c=1>
```

```
z <- caddmodN(c=c, bits=bits, c1=c1, c2=c2, a=a, x=z, y=6, N=N) # 4 + 6 mod 5
z
```

```
   ( 1 )    * |0>|a=0>|c2=0>|c1=0>|c=1>
```

## Controlled Multiplier modulo $N$

Now, like the `mult` function above a version performing $|x\rangle \rightarrow |x + y \mod N\rangle$. Here we also include the un-computation of the second register. `reg1` is the result register.

```
cmultmodN <- function(c, reg1, reg2, ancillas, x, y, N) {
  stopifnot(length(reg1) == length(reg2))
  ## need 4 ancilla registers
  stopifnot(length(ancillas) == 4 &&
            length(unique(ancillas)) == 4)
  n <- length(reg2)
  ## precompute terms in the sum
  s <- summands(y, N, n)
  ## start with |x>|0>
  for(i in c(1:n)) {
    x <- CCNOT(c(c, reg1[i], ancillas[4])) * x
    x <- caddmodN(c=ancillas[4], bits=reg2,
                  c1=ancillas[1], c2=ancillas[2],
                  a=ancillas[3],
                  x=x, y=s[i], N=N)
    x <- CCNOT(c(c, reg1[i], ancillas[4])) * x
  }
  ## now |x>|xy mod N>
  for(i in c(1:n)) {
    x <- CSWAP(c(c, reg1[i], reg2[i])) * x
  }
  ## now |xy mod N>|x>
  ## -y_inv mod N
  yinv <- N - moduloinverse(a=y, n=N)
  s <- summands(yinv, N, n)
  for(i in c(1:n)) {
    x <- CCNOT(c(c, reg1[i], ancillas[4])) * x
    x <- caddmodN(c=ancillas[4], bits=reg2,
                  c1=ancillas[1], c2=ancillas[2],
```

```
                  a=ancillas[3],
                  x=x, y=s[i], N=N)
    x <- CCNOT(c(c, reg1[i], ancillas[4])) * x
  }
  ## finally |xy mod N>|0>
  return(invisible(x))
}
```

For the un-computation of the second register, we start in the state (only the two registers `reg1` and `reg2`)

$$|x \cdot y \mod N\rangle|x\rangle\,.$$

Now we determine $yy_{\text{inv}} = 1 \mod N$ the modular inverse of $y$ (which is only possible, if $y$ and $N$ are co-prime). If we set $y' = xy \mod N$ it follows $y_{\text{inv}}y' = y_{\text{inv}}yx \mod N = x \mod N$. Thus, $x = y_{\text{inv}}y' \mod N$. So, if we perform the following trafo

$$|y'\rangle|x\rangle \to |y'\rangle|x - y_{\text{inv}}y' \mod N\rangle = |y'\rangle|0\rangle\,.$$

we obtain $|x \cdot y \mod N\rangle|0\rangle$.

Example with two 3 qubit registers, which is starting to become slow, because in total we need 11 qubits

```
basis <- c()
for(i in c(0:(2^11-1))) {
  basis[i + 1] <-
    paste0("|reg1=", i %/% (32*2^3) , ">|reg2=", (i %/% 32) %% 2^3 ,
           "|anc=", (i %/% 16) %% 2,
           (i %/% 8) %% 2, (i %/% 4) %% 2,
           (i%/%2) %% 2, ">|c=", i%%2, ">")
}
x <- CNOT(c(1,10)) * (H(1)*qstate(11, basis=basis))
x
```

```
   ( 0.7071068 )    * |reg1=0>|reg2=0|anc=0000>|c=0>
 + ( 0.7071068 )    * |reg1=2>|reg2=0|anc=0000>|c=1>
```

```
c <- 1
ancillas <- c(2:5)
reg2 <- c(6:8)
reg1 <- c(9:11)
N <- 5
z <- cmultmodN(c=c, reg1=reg1, reg2=reg2,
               ancillas=ancillas, x=x, y=3, N=N)
z
```

```
   ( 0.7071068 )    * |reg1=0>|reg2=0|anc=0000>|c=0>
 + ( 0.7071068 )    * |reg1=1>|reg2=0|anc=0000>|c=1>
```

```
z <- cmultmodN(c=c, reg1=reg1, reg2=reg2,
               ancillas=ancillas, x=z, y=3, N=N)
z
```

```
   ( 0.7071068 )    * |reg1=0>|reg2=0|anc=0000>|c=0>
 + ( 0.7071068 )    * |reg1=3>|reg2=0|anc=0000>|c=1>
```

```
z <- cmultmodN(c=c, reg1=reg1, reg2=reg2,
               ancillas=ancillas, x=z, y=3, N=N)
z
```

```
   ( 0.7071068 )    * |reg1=0>|reg2=0|anc=0000>|c=0>
 + ( 0.7071068 )    * |reg1=4>|reg2=0|anc=0000>|c=1>
```

```
z <- cmultmodN(c=c, reg1=reg1, reg2=reg2,
               ancillas=ancillas, x=z, y=3, N=N)
z
```

```
   ( 0.7071068 )    * |reg1=0>|reg2=0|anc=0000>|c=0>
 + ( 0.7071068 )    * |reg1=2>|reg2=0|anc=0000>|c=1>
```

This seems to work up to this point.

## Exponentiation modulo $N$

For the order finding algorithm, we have to implement the operation $f_{a,N}(x) = a^x y \mod N$, where we set $y = 1$ in the following. All this is stored in a $n$ qubit register with $2^n > N$.

So, the following function implements the unitary operation naively

$$|x\rangle \rightarrow |xy^a \mod N\rangle$$

```
cexpomodN <- function(c, reg1, reg2, ancillas, x, y, a, N) {
  stopifnot(length(reg1) == length(reg2))
  ## need 4 ancilla registers
  stopifnot(length(ancillas) == 4 &&
            length(unique(ancillas)) == 4)
  for(i in c(1:a)) {
    x <- cmultmodN(c=c, reg1=reg1, reg2=reg2,
                   ancillas=ancillas, x=x, y=y, N=N)
  }
  return(invisible(x))
}
```

Example, performing the same example operation as above, i.e. starting with $x = 2$ and multiplying it with $y^a \mod N$, with $a = 4$, $y = 3$ and $N = 5$.

```
x <- CNOT(c(1,10)) * (H(1)*qstate(11, basis=basis))
x
```

```
   ( 0.7071068 )    * |reg1=0>|reg2=0|anc=0000>|c=0>
 + ( 0.7071068 )    * |reg1=2>|reg2=0|anc=0000>|c=1>
```

```
x <- cexpomodN(c, reg1, reg2, ancillas, x, y=3, a=4, N)
x
```

```
   ( 0.7071068 )    * |reg1=0>|reg2=0|anc=0000>|c=0>
 + ( 0.7071068 )    * |reg1=2>|reg2=0|anc=0000>|c=1>
```

The result should equal 2.

Using the binary representation of $x$, we can write

$$y^a = y^{2^0 a_0} \cdot y^{2^1 a_1} \cdot \ldots y^{2^{m-1} a_{m-1}}$$

if $x$ is stored with $m$ bits. So, alternatively, if we start with the result register in state $|1\rangle$ we have to multiply successivly $m$-times by $a^{2^i} \mod n$ depending on the value of the qubit $|x_i\rangle$. This task can be also achieved with the controlled multiplier developed above. However, we need one more register to store $a$. Let's cheat here a bit and use a classical `if`-statement. For large $a$ this implementation is of course much faster, but the factors $y^{2^i}$ become very large very quickly. That's why we apply the modulo

```
cexpomodN2 <- function(c, reg1, reg2, ancillas, x, y, a, N) {
  stopifnot(length(reg1) == length(reg2))
  ## need 4 ancilla registers
  stopifnot(length(ancillas) == 4 &&
            length(unique(ancillas)) == 4)
  ab <- as.integer(intToBits(a))
  n <- max(which(ab == 1))
  y2 <- y %% N
  for(i in c(1:n)) {
    if(ab[i] == 1) {
      x <- cmultmodN(c=c, reg1=reg1, reg2=reg2,
                     ancillas=ancillas, x=x, y=y2, N=N)
    }
    y2 <- ((y2%%N) * (y2%%N)) %% N # y2=y^(2^i) mod N
  }
  return(invisible(x))
}
```

Example

```
x <- CNOT(c(1,10)) * (H(1)*qstate(11, basis=basis))
x
```

```
   ( 0.7071068 )    * |reg1=0>|reg2=0|anc=0000>|c=0>
 + ( 0.7071068 )    * |reg1=2>|reg2=0|anc=0000>|c=1>
```

```
x <- cexpomodN2(c, reg1, reg2, ancillas, x, y=3, a=4, N)
x
```

```
   ( 0.7071068 )    * |reg1=0>|reg2=0|anc=0000>|c=0>
 + ( 0.7071068 )    * |reg1=2>|reg2=0|anc=0000>|c=1>
```

# References


Draper, Thomas G. 2000. "Addition on a Quantum Computer." *arXiv Preprint Quant-Ph/0008033*.

Vedral, Vlatko, Adriano Barenco, and Artur Ekert. 1996. "Quantum Networks for Elementary Arithmetic Operations." *Physical Review A* 54 (1): 147–53. https://doi.org/10.1103/physreva.54.147.