Artificial intelligence

technical report

Author:
A. C. LISBOA, PhD

2026-01-31

Updates

19/12/2025 Adriano Chaves Lisboa

i. vector modular artificial intelligence math reasonably stable;
ii. implementation very uninstable.

01/01/2026 Adriano Chaves Lisboa

i. vector modular artificial intelligence math stable;
ii. Python implementation stable.

23/01/2026 Adriano Chaves Lisboa

i. vector modular artificial intelligence math stable;
ii. Matlab implementation stable.

Contents

1 Introduction

First principles approach.

2 Artificial intelligence

2.1 Model

Artificial intelligence may be modeled in general as a function

$$y = f(w, x), \quad x \in \mathbb{X}, w \in \mathbb{W}, y \in \mathbb{Y} \tag{1}$$

that allows mapping input x (in set $\mathbb{X}$) to output y (in set $\mathbb{Y}$) after being trained to obtain parameters w (in set $\mathbb{W}$) that minimize errors over a sample set of S input-output pairs $(\hat{x}_s, \hat{y}_s) \in \mathbb{S}$, $s = 0, ..., S-1$. For instance, a linear parameterization would be $y = w_0 + w_1 x = [1 \;\; x]w, x, y \in \mathbb{R}, w \in \mathbb{R}^2$.

2.2 Training

Training is an optimization problem in the form

$$w = \arg\min_{w} \;\; d(f(w, \hat{x}_k), \hat{y}_k) \tag{2}$$

where $d(y, \hat{y}) : \mathbb{Y} \times \mathbb{Y} \to \mathbb{R}$ is a distance measure between $y \in \mathbb{Y}$ and $\hat{y} \in \mathbb{Y}$.

3 Tight universal parameterization

Definition 1 (consistent sample). *A sample set $\mathbb{S}$ is consistent if the input samples $\hat{x}$ (where $(\hat{x}_s, \hat{y}_s) \in \mathbb{S}$) are unique (i.e. $\hat{x}_s \neq \hat{x}_{s'}, \forall s \neq s'$).*

A consistent sample set $\mathbb{S}$ implies $|\mathbb{Y}|, |\mathbb{S}| \leq |\mathbb{X}|$.

Definition 2 (universal). *A parameterization is universal when there is always a set of parameters to achieve any input-output combination of consistent sample set $\mathbb{S}$.*

Definition 3 (tight). *A parameterization is tight when the cardinality of parameter space equals to the cardinality of input-output combinations, i.e. when $|\mathbb{W}| = |\mathbb{Y}|^{|\mathbb{X}|}$.*

3.1 Basic problem statement

When universality and tightness are the analysis focus, and also finite data, a unique correspondence between input and output, and also the whole set of their combinations and a tight parameter space to represent them, naturally lead to integer numbers due to finiteness both in theory and in practice. Indeed, in practice, a continuous number is typically approximated by an integer mantissa and exponent representation. So, the most basic problem to be analyzed is based on integer set maps.

Given an input $x \in \{0, ..., X-1\}$ and an output $y \in \{0, ..., Y-1\}$, how to define a tight universal parameterization capable of learning any consistent training set $(\hat{x}_s, \hat{y}_s)$, $s = 0, ..., S-1$?

There are Y^X possible input-output combinations, so that a universal parameterization must have at least this amount of combinations in parameter space. Is it possible to define a parameterization tight to this bound?

3.2 Modular polynomial

The modular polynomial parameterization is defined by

$$y = \sum_{i=0}^{X-1} w_i x^i \mod Y \tag{3}$$

$$= [1 \;\; x \;\; x^2 \; ... \; x^{X-1}] w \mod Y \tag{4}$$

$$= P_x w \mod Y \tag{5}$$

where the parameters $w \in \mathbb{W}$ are tight (i.e. $|\mathbb{W}| = Y^X$) and they are universal only for $X = Y = M$ prime or a power of a prime (i.e. if the field is finite).

3.2.1 Training

The training of the modular polynomial parameterization model for a consistent sample set is exact and given by the solution of a system of modular linear equations given by

$$w^\star = \begin{bmatrix} 1 & \hat{x}_0 & \hat{x}_0^2 & ... & \hat{x}_0^{M-1} \\ 1 & \hat{x}_1 & \hat{x}_1^2 & ... & \hat{x}_1^{M-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \hat{x}_{S-1} & \hat{x}_{S-1}^2 & ... & \hat{x}_{S-1}^{M-1} \end{bmatrix}^{-1} \begin{bmatrix} \hat{y}_0 \\ \hat{y}_1 \\ \vdots \\ \hat{y}_{S-1} \end{bmatrix} \mod M \tag{6}$$

$$= P_x^{-1} \hat{y} \mod M \tag{7}$$

where P_x is the polynomial space and for $S < M$ (e.g. a consistent sample set) the solution is optimal on the least square error measure $d = (P_x w - \hat{y})^T (P_x w - \hat{y})$ using the

pseudo-inverse defined by

$$(P_x^T P_x)^{-1} P_x^T \tag{8}$$

Since solving a system of equations involve divisions, the modulus M must be a prime to allow training. Furthermore, when $M > S$, the last $M - S$ elements of w are null, so that it is enough to consider the first S terms of the M modular polynomial and solve a square $S \times S$ system of linear equations.

3.2.1.1 Complexity analysis For a consistent sample set, the training time complexity is given by the solution of the $S \times S$ modular linear equation system (see Figure 1 for training timing in practice), for $S \leq M$, i.e. $O(S^3)$, and the storage complexity is given by the matrix P_x as $O(S^2 \log M)$ bytes. To store the parameters, at most S non-null values in modular M algebra are necessary, i.e. $O(S \log M)$ bytes of storage.

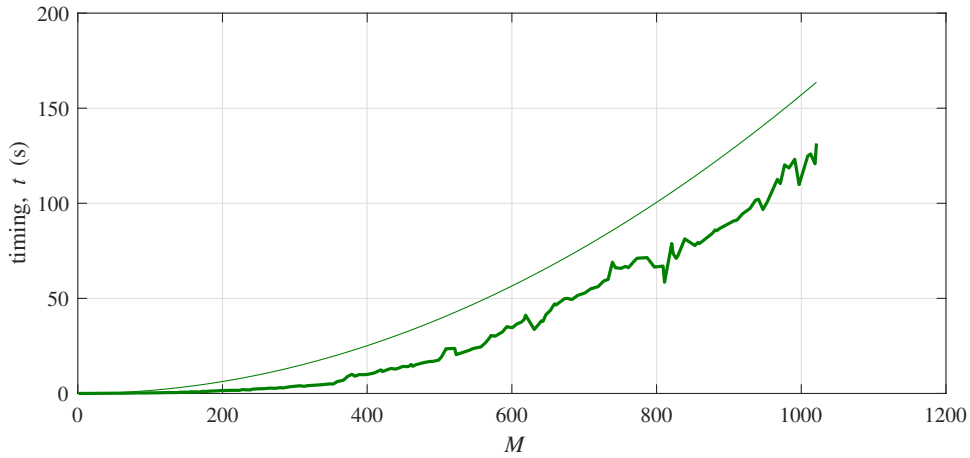

Figure 1: Average training timing for 4 repetitions on random datasets. Fits on $t \leq 157 \times 10^{-6} M^2$ [s].

Considering that $b = \log_{2^8} M$ bytes are needed to store a M modular residue and that a computer has B bytes of available memory, the problem size is limited by

$$K^2 \log_{2^8} M \leq B \tag{9}$$

to be properly trained (e.g. $S = 2^{18} = 262,144$ samples using modulus $M = 2^{64}$ would require at least 2^{44}B≈ 18TB of memory, which is currently infeasible to train, or 2^{36}B≈ 69GB of memory for $K = 2^{16} = 65,536$ samples modulus $M = 2^{32}$ requiring about some hours of processing training in a modern supercomputer).

3.2.1.2 Continuous input and output Any continuous input $\hat{x} \in [\underline{\hat{x}}, \bar{\hat{x}}]$ can be mapped into $\{0, ..., M-1\}$ using

$$x = \left\lfloor \frac{\hat{x} - \underline{\hat{x}}}{\bar{\hat{x}} - \underline{\hat{x}}} \frac{M}{1+\epsilon} \right\rfloor \mod M \tag{10}$$

and, similarly, any continuous output $y \in [\underline{\hat{y}}, \bar{\hat{y}}]$ can be mapped into $\{0, ..., M-1\}$ using

$$y = \left\lfloor \frac{\hat{y} - \underline{\hat{y}}}{\bar{\hat{y}} - \underline{\hat{y}}} \frac{M}{1+\epsilon} \right\rfloor \mod M \tag{11}$$

where ϵ is an infinitesimal number, which in practice is the machine floating point representation error.

3.2.1.3 Recurrent input and output The output of a first entry may be the input in the next time (i.e. a recurrent artificial intelligence)

$$x_{t+1} = y(x_t) \tag{12}$$

for a given x_0, so that a sequence of up to M unique values may be obtained until a loop is reached.

3.2.1.4 Vector input and output A vector of n modular M variables $\hat{x}$ may be converted to a single variable using scaling and shifting

$$x = \sum_{i=1}^{n} \left\lfloor \frac{\hat{x}_i}{M-1} \frac{\hat{X}}{1+\epsilon} \right\rfloor \hat{X}^{i-1} \mod M, \quad \hat{X} = \left\lfloor \sqrt[n]{M} \right\rfloor \tag{13}$$

for concatenation of variables (i.e. $\hat{x} = [0 \;\; 0 \;\; 0...]$ implies $x = 0$, ... $\hat{x} = [(M-1) \;\; 0 \;\; 0...]$ implies $x = \hat{X} - 1$, $\hat{x} = [0 \;\; (M-1) \;\; 0...]$ implies $x = (\hat{X} - 1)\hat{X}$, ...), where ϵ is an infinitesimal number, which is not necessarily universal or tight anymore; or without scaling (e.g. variables may already be in the correct range) considering $\hat{x}_i \in \{0, ..., \hat{X}_i - 1\}$

$$x = \sum_{i=1}^{n} \hat{x}_i \prod_{j=1}^{i-1} \hat{X}_j \mod M \tag{14}$$

for concatenation of variables (i.e. $\hat{x} = [0 \;\; 0 \;\; 0...]$ implies $x = 0$, ... $\hat{x} = [(M-1) \;\; 0 \;\; 0...]$ implies $x = \hat{X}_1 - 1$, $\hat{x} = [0 \;\; (M-1) \;\; 0...]$ implies $x = (\hat{X}_2 - 1)\hat{X}_1$, ...), which is still universal as long as $\prod_{i=1}^{n} \hat{X}_i \leq M$ and tight as long as $\prod_{i=1}^{n} \hat{X}_i = M$.

3.2.1.5 Vector modular training Algorithm 1 depicts how to train a N-vector modular artificial intelligence. Data exposure for inference and performance improvements are omitted for simplicity. To cope with multiple inputs, each input is scaled to a specific range in X and then combined to other inputs like they were indexes of a N-dimensional matrix being converted to a single index related to elements position in memory. The information lost during scaling is refined in further layers, as shown in Figure 2. This approach tackles most significant parts of the number first and the number is refined as needed to not lose information.

Algorithm 1 Training of a vector modular artificial intelligence.

Input

$(\hat{x}_s, \hat{y}_s) \in \mathbb{S}$ input $\hat{x}_s \in \{0, ..., M-1\}^N$ output $\hat{y}_s \in \{0, ..., M-1\}$, $s = 0, ..., S-1$, sample set

$G \in \mathbb{N}$ input group size in $\{2, 3, 4..., \lfloor \log_2 M \rfloor\}$

Output

$w \in \mathbb{W}$ trained parameters

$\Delta y \in \mathbb{N}^S$ training error

$\mathbb{X}_0 \leftarrow \bigcup_s \{\hat{x}_s\}$ ▷ starting input set

$\ell \leftarrow 0$ ▷ layer counter

$\bar{\ell} \leftarrow \log_G(M-1)$ ▷ maximum refinement layer

$\Delta y \leftarrow \hat{y}$ ▷ starting output error

$\mathbb{W} \leftarrow \emptyset$ ▷ starting parameter set

while $\mathbb{X}_\ell \neq \emptyset$ **do**

$\mathbb{G}_\ell \leftarrow \begin{cases} \text{every combination of } |\mathbb{X}_\ell| \text{ choose } G, & G < |\mathbb{X}_\ell| \\ \{(0, ..., |\mathbb{X}_\ell| - 1)\}, & \text{otherwise} \end{cases}$ ▷ input groups

$\mathbb{X}_{\ell+1} \leftarrow \begin{cases} \bigcup_{\hat{x}_s \in \mathbb{X}_\ell} \{\hat{x}_s G \mod M\}, & \ell < \bar{\ell} \text{ and } n > 1 \\ \emptyset, & \text{otherwise} \end{cases}$ ▷ refined input

for $n \in \mathbb{G}_\ell$, $n \in \{0, ..., |\mathbb{X}_\ell|\}^{\bar{g}}$, $\bar{g} = \min\{G, |\mathbb{X}_\ell|\}$ **do**

$X_g \leftarrow \left\lfloor \frac{M}{\bar{g}} \right\rfloor + \begin{cases} 0, & g \geq (M \mod \bar{g}) \\ 1, & \text{otherwise} \end{cases}, \forall g = 0, ..., \bar{g} - 1$ ▷ input ranges

$x_s \leftarrow \sum_{g=0}^{\bar{g}-1} \left\lfloor \frac{\hat{x}_{s,n_g}}{M-1} \frac{X_g}{1+\epsilon} \right\rfloor \prod_{g'=0}^{g-1} X_{g'}, \forall s$ ▷ input grouping (14)

$P_{s,e} \leftarrow x_s^e \mod M, \forall s, e = 0, ..., S-1$ ▷ polynomial space (6)

$\Delta y'_s \leftarrow \min_{s' \,:\, x_{s'} = x_s} \Delta y_{s'}, \forall s$ ▷ consistent error, skip training if $\Delta y' = 0$

$w \leftarrow P^{-1} \Delta y' \mod M$ ▷ incremental training (7)

$\Delta y \leftarrow \Delta y - Pw$ ▷ output error reduction, stop if $\Delta y = 0$

$\mathbb{W} \leftarrow \mathbb{W} \bigcup \{w\}$ ▷ append parameters

end for

$\ell \leftarrow \ell + 1$ ▷ increment layer

end while

Algorithm 2 allows to train with a reduced modulus in order to avoid Incomplete input and output errors. The refinement strategy (shown in Figure 2) is analogous to the one used in the vector modular training Algorithm 1: cope with the most significant parts first and then refine as needed. In order to take maximum advantage of this strategy, the input combination of vector training must be such that lower refined inputs are treated first.

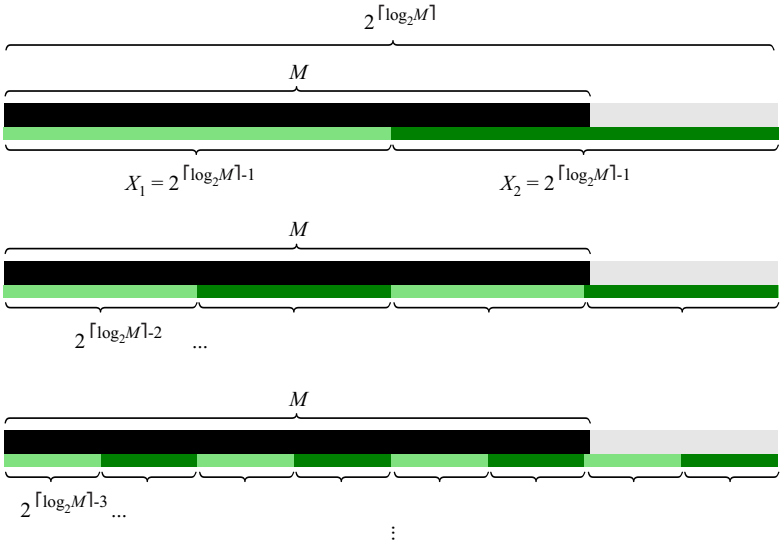

Figure 2: Number refinement for $M' = 2$. In order to keep the number in the correct range, either i. the number is scaled to a proper specific range X (see line 11 in Algorithm 1), or ii. the M' power next to M $\bar{r}$-remainder ($\bar{r} = \lceil \log_{M'} M \rceil$) of the number multiplied by M'^{r} is divided by $M'^{\bar{r}-1}$ (see line 3 in Algorithm 2).

Algorithm 2 Fixed modulus training of a vector modular artificial intelligence.

Input

$(\hat{x}_s, \hat{y}_s) \in \mathbb{S}$	input $\hat{x}_s \in \{0, ..., M-1\}^N$ output $\hat{y}_s \in \{0, ..., M-1\}$, $s = 0, ..., S-1$, sample set
$G \in \mathbb{N}$	input group size in $\{2, 3, 4..., \lfloor \log_2 M \rfloor\}$
$M' \in \mathbb{N}$	fixed modulus, a prime number $M' \leq M$

Output

$w \in \mathbb{W}$	trained parameters
$\Delta y \in \mathbb{N}^S$	training error

1: $\hat{x}' \leftarrow []$ ▷ starting refined input

2: **for** $r = 0, 1, ..., \lceil \log_{M'} M \rceil - 1$ **do**

3: $\hat{x}' \leftarrow \left[\hat{x}', \left\lfloor \frac{\hat{x} M'^{r} \mod M'^{\lceil \log_{M'} M \rceil}}{M'^{\lceil \log_{M'} M \rceil - 1}} \right\rfloor\right]$ ▷ refined dataset

4: **end for**

5: $\mathbb{W}, \Delta y \leftarrow \mathrm{MAI}\left(\bigcup_s \{(\hat{x}'_s, \hat{y}_s)\}, G\right)$ ▷ training (Algorithm 1)

3.2.1.6 Efficient vector modular training Given $\hat{x} \in \{0, ..., X-1\}^{S \times N}$ and $\hat{y} \in \{0, ..., Y-1\}^S$, find a group $n \in \{0, ..., N-1\}^G$, $G \leq N$, such that $\hat{y}' \neq 0$, where $\hat{y}'_s = \min_{s' \,:\, x_s = x_{s'}} \hat{y}_{s'}$, $x_s = \sum_{g=0}^{G-1} \hat{x}_{s,n_g} X^g$, exploring as few combinations as possible among the possible N choose G.

3.2.1.7 Optimal vector architecture

The number size in a G-group N-vector M-modular artificial intelligence of is given by

$$W = M^U, U \leq \sum_{r=0}^{\lfloor \log_G(M-1) \rfloor} u_r, u_{r+1} = \frac{u_r}{2}(u_r + 3), u_0 = N \tag{15}$$

so that the higher N and M are, the higher W is, and the higher G is, the lower W is.

3.2.1.8 Tensor input and output

For N-dimensional tensor input variables $x_{i'}$ index by $i'_c \in \{1, ..., m_c\}$, $c = 1, ..., N$, using Einstein notation, a modular polynomial convolution parameterization

$$y_j = \sum_{m=0}^{M-1} w_{i,m} x^m_{i'(i,j)} \mod M \tag{16}$$

where

$$i'(i_c, j_c) = j_c + i_c - 1, \quad i_c \in \{1, ..., n_c\}, j_c \in \{1, ..., m_c - n_c + 1\}, c \in \{1, ..., N\} \tag{17}$$

may be a good choice for reducing the number of parameters or output size, but the parameterization becomes not universal and not tight. Notice that when $n_c = m_c$, $\forall c$, the modular polynomial convolution parameterization degenerates into the conventional modular polynomial parameterization.

3.2.1.9 Uncertain input and output

An exact training may be challenging to avoid overfitting to noisy data, i.e. uncertain input and output, as shown in Figure 3.

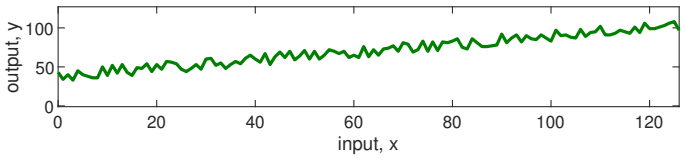

Figure 3: An exact model follows whatever the training data is (complete data for $\hat{x} \in \{0, 1, ..., 126\}$).

3.2.1.10 Incomplete input and output

Interpolation and extrapolation are also a challenge, as shown in Figure 4.

3.2.1.11 Round off errors

3.2.2 Examples

3.2.2.1 Binary input and output

For $x, y \in \{0, 1\}$ a universal parameterization would be

$$y = w_0 + w_1 x \mod 2, \quad w_1 \in \{0, 1\}, w_0 \in \{0, 1\} \tag{18}$$

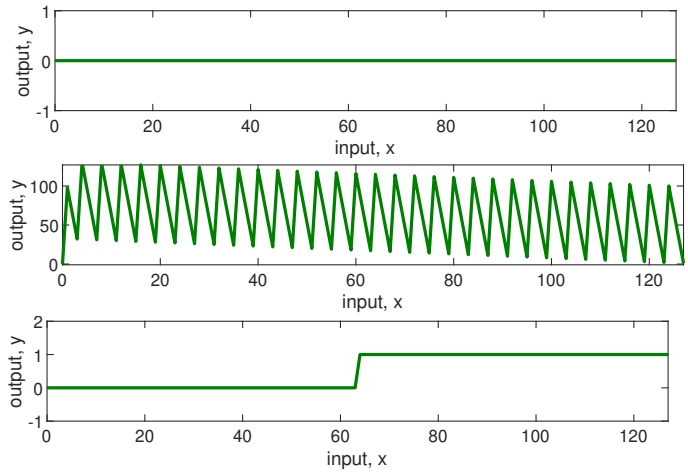

Figure 4: A 0-parameter model for $\hat{x} = (0, 127)$ and $\hat{y} = (0, 0)$ (top), and a 2-parameter model for $\hat{x} = (0, 127)$ and $\hat{y} = (0, 1)$ (center) reveals the sensitivity and interpolation problems, which is solved with a low fixed modulus $M = 2$ (bottom).

so that

i. $w = [0, 0]$ for $\hat{x} = [0, 1]$ and $\hat{y} = [0, 0]$;

ii. $w = [0, 1]$ for $\hat{x} = [0, 1]$ and $\hat{y} = [0, 1]$;

iii. $w = [1, 1]$ for $\hat{x} = [0, 1]$ and $\hat{y} = [1, 0]$;

iv. $w = [1, 0]$ for $\hat{x} = [0, 1]$ and $\hat{y} = [1, 1]$;

and the parameterization is universal and tight.

3.2.2.2 Ternary input and output

For $x, y \in \{0, 1, 2\}$ a universal parameterization would be

$$y = w_0 + w_1 x + w_2 x^2 \mod 3, \quad w_2 \in \{0, 1, 2\}, w_1 \in \{0, 1, 2\}, w_0 \in \{0, 1, 2\} \tag{19}$$

so that

i. $w = [0, 0, 0]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [0, 0, 0]$;

ii. $w = [0, 1, 2]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [0, 0, 1]$;

iii. $w = [0, 2, 1]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [0, 0, 2]$;

iv. $w = [0, 2, 2]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [0, 1, 0]$;

v. $w = [0, 0, 1]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [0, 1, 1]$;

vi. $w = [0, 1, 0]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [0, 1, 2]$;

vii. $w = [0, 1, 1]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [0, 2, 0]$;

viii. $w = [0, 2, 0]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [0, 2, 1]$;

ix. $w = [0, 0, 2]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [0, 2, 2]$;

x. $w = [1, 0, 2]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [1, 0, 0]$;

xi. $w = [1, 1, 1]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [1, 0, 1]$;

xii. $w = [1, 2, 0]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [1, 0, 2]$;

xiii. $w = [1, 2, 1]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [1, 1, 0]$;

xiv. $w = [1, 0, 0]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [1, 1, 1]$;

xv. $w = [1, 1, 2]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [1, 1, 2]$;

xvi. $w = [1, 1, 0]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [1, 2, 0]$;

xvii. $w = [1, 2, 2]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [1, 2, 1]$;

xviii. $w = [1, 0, 1]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [1, 2, 2]$;

xix. $w = [2, 0, 1]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [2, 0, 0]$;

xx. $w = [2, 1, 0]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [2, 0, 1]$;

xxi. $w = [2, 2, 2]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [2, 0, 2]$;

xxii. $w = [2, 2, 0]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [2, 1, 0]$;

xxiii. $w = [2, 0, 2]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [2, 1, 1]$;

xxiv. $w = [2, 1, 1]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [2, 1, 2]$;

xxv. $w = [2, 1, 2]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [2, 2, 0]$;

xxvi. $w = [2, 2, 1]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [2, 2, 1]$;

xxvii. $w = [2, 0, 0]$ for $\hat{x} = [0, 1, 2]$ and $\hat{y} = [2, 2, 2]$;

and the parameterization is universal and tight.

4 Case studies

4.1 Fire detection

4.1.1 Detection

4.1.2 Segmentation

4.2 Bus inspection

4.2.1 Nut angle

4.3 First language models

i. concepts development based on physical world and its measures;

ii. connections between concepts to create new concepts and reduce amount of information;

iii. switch flips.

4.3.1 Brazil

4.3.2 Brazil and United States

4.3.3 Brazil, United States, Spain and Israel

4.4 Psychology of discourse

A Code

A.1 Modular artificial intelligence

Modular algebra can be conveniently implemented by a container for residue or congruence class, overloading operators in order to use conventional algorithms. Most of operators are simply common algebra with a remainder at the end, with a notable exception for division using Euclidean algorithm and a careful power operator to avoid overflow.

Many spaces are created to allow efficient training and tests, including truth space, polynomial space, and interleaved polynomial space.

The training comprises a simple but hard to code algorithm.

A.1.1 Python

```
# first principles
# test driven development
# criterion: self contained coded
# clue: Grok has accuratly more detailed algorithms

'''References:
[1] A. C. Lisboa. "Artificial intelligence", technical report, Gaia, gaiasd.com/AI.pdf, 2026.
'''

import numpy as np
import sys
import io
import time
import json
import copy

np.set_printoptions(threshold=sys.maxsize)
np.set_printoptions(precision=0, suppress=True)

# modular algebra
class modular_residue:
   # Construction.
   def __init__(self, r, M, b=None):
      ''' Construct a modular residue number.

      Args:
         r (numpy array of int): residue
         M (int): modulus
         b (bool, optional): modulus is prime indicator (default is a prime predicate)

      Returns:
         (modular_residue): a modular residue number
      '''

      # Setup.
      self._modulus = M
      self._residue = r
      self._is_prime = is_prime(M) if b is None else b

   # Read only attributes.
   def residue(self):
      return self._residue

   def dtype(self):
      return str(self._residue.dtype)

   def modulus(self):
      return self._modulus

   def is_prime(self):
      return self._is_prime

   # Operators.
   def __setattr__(self,name,value):
      # Error check.
      if name == '_residue':
         if not isinstance(value,np.ndarray) or str(value.dtype) not in ['int8', 'int16', 'int32', 'int64']
```

```
            raise ValueError('Type for residue must be signed integer')
         value %= self._modulus
      elif name == '_modulus':
         if not isinstance(value,int):
            raise ValueError('Modulus must be an integer')
      elif name == '_is_prime':
         if not isinstance(value,bool):
            raise ValueError('Is prime indicator must be a boolean')
      else:
         raise ValueError('Non existing attribute name \'{}\''.format(name))

      # Setup.
      super().__setattr__(name,value)

   def __add__(self, other):
      # Error check.
      if isinstance(self,modular_residue) and isinstance(other,modular_residue) and self._modulus != other._modul
         raise ValueError('Operand modulus must be the same')

      # Extract info.
      bp = self._is_prime if isinstance(self,modular_residue) else other._is_prime # is prime indicator
      a = self._residue if isinstance(self,modular_residue) else self # first operand
      b = other._residue if isinstance(other,modular_residue) else other # second operand
      M = self._modulus if isinstance(self,modular_residue) else other._modulus # modulus

      # Modular sum.
      return modular_residue(a+b, M, bp)

   def __radd__(self,other):
      return self.__add__(other)

   def __neg__(self):
      return modular_residue(self._modulus-self._residue, self._modulus, self._is_prime)

   def __sub__(self, other):
      # Error check.
      if isinstance(other,modular_residue) and self._modulus != other._modulus:
         raise ValueError('Operand modulus must be the same.')

      # Extract info.
      bp = self._is_prime if isinstance(self,modular_residue) else other._is_prime # is prime indicator
      a = self._residue if isinstance(self,modular_residue) else self # first operand
      b = other._residue if isinstance(other,modular_residue) else other # second operand
      M = self._modulus if isinstance(self,modular_residue) else other._modulus # modulus

      # Modular sum.
      return modular_residue(a+M-b, M, bp)

   def __rsub__(self,other):
      return -(self.__sub__(other)) # there is some switch flip in this overload...

   def __mul__(self, other):
      # Error check.
      if isinstance(other,modular_residue) and self._modulus != other._modulus:
         raise ValueError('Operand modulus must be the same')

      # Extract info.
      bp = self._is_prime if isinstance(self,modular_residue) else other._is_prime # is prime indicator
      a = self._residue if isinstance(self,modular_residue) else self # first operand
      b = other._residue if isinstance(other,modular_residue) else other # second operand
      M = self._modulus if isinstance(self,modular_residue) else other._modulus # modulus

      # Modular multiplication.
```

```
        return modular_residue(a*b, M, bp)

    def __rmul__(self, other):
        return self*other

    def __matmul__(self, other):
        # Error check.
        if isinstance(other,modular_residue) and self._modulus != other._modulus:
            raise ValueError('Operand modulus must be the same')
        if len(self._residue.shape) != 2 or len(other._residue.shape) != 2:
            raise ValueError('Operands must be matrices')

        # Extract info.
        bp = self._is_prime if isinstance(self,modular_residue) else other._is_prime # is prime indicator
        a = self._residue if isinstance(self,modular_residue) else self # first operand
        b = other._residue if isinstance(other,modular_residue) else other # second operand
        M = self._modulus if isinstance(self,modular_residue) else other._modulus # modulus

        # Modular multiplication.
        return modular_residue(a@b, M, bp)

    def __mod__(self, other):
        # Error check.
        if not isinstance(self,modular_residue):
            raise TypeError('Operand must be modular residues')
        if not isinstance(other,int):
            raise TypeError('Modulus must be an integer')

        # Extract info.
        a = self._residue # residue
        M = other # modulus

        # New modulus
        return modular_residue(a%M, M, is_prime(M))

    def __pow__(self, other):
        # Error check.
        if isinstance(other,modular_residue) and self._modulus != other._modulus:
            raise ValueError('Operand modulus must be the same')
        if isinstance(other,np.ndarray) and other.dtype != self._residue.dtype:
            other.astype(self._residue.dtype)

        # Extract info.
        bp = self._is_prime if isinstance(self,modular_residue) else other._is_prime # is prime indicator
        a = self._residue if isinstance(self,modular_residue) else self # first operand
        b = other._residue if isinstance(other,modular_residue) else other # second operand
        M = self._modulus if isinstance(self,modular_residue) else other._modulus # modulus
        s = a.shape if not isinstance(b,np.ndarray) or sum(a.shape) > sum(b.shape) else b.shape # shape

        # Efficient power.
        if M==1:
            r = np.zeros(s)
        elif self._is_prime:
            r = (a ** (b % (M-1))) % M
        else:
            r = np.ones(s,dtype=a.dtype) # power
            ar = (a%M)*np.ones(s,dtype=a.dtype)# power basis
            br = b*np.ones(s,dtype=a.dtype) # power exponent
            while np.any(br > 0):
                ib = np.where((br % 2) > 0)[0]
                r[ib] *= ar[ib]
                r[ib] %= M
                ar *= ar
```

```
            ar %= M
            br //= 2
        r[np.logical_and(a==0,b==0)] = 1
        r[np.logical_and(a==0,b!=0)] = 0
        return modular_residue(r, M, bp)

    def __rpow__(self, other):
        if isinstance(other,int):
            other = modular_residue(np.array([other],dtype=self._residue.dtype), self._modulus, self._is_prime)
        else:
            raise TypeError('Invalid basis type')
        return other.__pow__(self) # there is some switch flip in this overload...

    def __truediv__(self, other):
        # Error check.
        if isinstance(other,modular_residue) and self._modulus != other._modulus:
            raise ValueError('Operand modulus must be the same')
        if isinstance(other,np.ndarray) and other.dtype != self._residue.dtype:
            other.astype(self._residue.dtype)

        # Extract info.
        bp = self._is_prime if isinstance(self,modular_residue) else other._is_prime # is prime indicator
        a = self._residue if isinstance(self,modular_residue) else self # first operand
        b = other._residue if isinstance(other,modular_residue) else other # second operand
        M = self._modulus if isinstance(self,modular_residue) else other._modulus # modulus
        s = a.shape if not isinstance(b,np.ndarray) or sum(a.shape) > sum(b.shape) else b.shape # shape
        ns = len(s) # tensor dimension
        if a.shape != b.shape:
            st = np.array([1]*ns,dtype=int)
            for ins in range(ns):
                if a.shape[ins] != b.shape[ins]:
                    ds = np.array([1]*ns,dtype=int)
                    if a.shape[ins]==1:
                        ds[ins] = b.shape[ins]
                        a = np.tile(a,tuple(ds))
                    elif b.shape[ins]==1:
                        ds[ins] = a.shape[ins]
                        b = np.tile(b,tuple(ds))
                    else:
                        raise ValueError('Inconsistent operant sizes')
        dtype = a.dtype if isinstance(a,np.ndarray) else b.dtype
        n = np.prod(s)

        # Efficient divide: Euclidean algorithm.
        bm = np.vstack(((b.reshape(-1)%M).astype('int64'), np.tile(np.array([M],dtype='int64'),(n)))) # put in rang
        x = np.vstack((np.zeros((n),dtype='int64'),np.ones((n),dtype='int64')))
        iin = np.where(bm[1,:]!=0)[0]
        while sum(iin.shape) > 0:
            # Euclidean expansion.
            q = bm[0,iin] // bm[1,iin] # quotient
            bm[:,iin] = np.vstack((bm[1,iin], bm[0,iin]%bm[1,iin]))
            x[:,iin] = np.vstack((x[1,iin]-q*x[0,iin],x[0,iin]))
            iin = iin[bm[1,iin]!=0]

        # Error check.
        if M!=1 and np.any(np.logical_and(bm[0,:]!=1,b.reshape(-1)!=0)):
            raise ValueError('Divisor and modulus must be coprime')

        # Output arguments.
        return modular_residue((np.array([a],dtype=dtype) if isinstance(a,int) else a)*x[1,:].reshape(s), M, bp)

    def __rtruediv__(self, other):
        if isinstance(other,int):
```

```
            other = np.array([other],dtype=self._residue.dtype)
        if isinstance(other,np.ndarray):
            return modular_residue(other, self._modulus, self._is_prime) / self._residue # there is some switc
        else:
            raise TypeError('Not implemented operator for these operand types')

    def __floordiv__(self, other):
        return self.__truediv__(other)

    def __rfloordiv__(self, other):
        return self.__floordiv__(other) # there is some switch flip in this overload... and every non-commuta

    def __eq__(self, other):
        a = self._residue if isinstance(self,modular_residue) else self # first operand
        b = other._residue if isinstance(other,modular_residue) else other # second operand
        return a == b

    def __ne__(self, other):
        a = self._residue if isinstance(self,modular_residue) else self # first operand
        b = other._residue if isinstance(other,modular_residue) else other # second operand
        return a != b

    def __lt__(self, other):
        a = self._residue if isinstance(self,modular_residue) else self # first operand
        b = other._residue if isinstance(other,modular_residue) else other # second operand
        return a < b

    def __le__(self, other):
        a = self._residue if isinstance(self,modular_residue) else self # first operand
        b = other._residue if isinstance(other,modular_residue) else other # second operand
        return a <= b

    def __gt__(self, other):
        a = self._residue if isinstance(self,modular_residue) else self # first operand
        b = other._residue if isinstance(other,modular_residue) else other # second operand
        return a > b

    def __ge__(self, other):
        a = self._residue if isinstance(self,modular_residue) else self # first operand
        b = other._residue if isinstance(other,modular_residue) else other # second operand
        return a >= b

    def __getitem__(self, key):
        return modular_residue(self._residue[key], self._modulus, self._is_prime)

    def __setitem__(self, key, value):
        value = value._residue if isinstance(value,modular_residue) else value
        self._residue[key] = value

    def __str__(self):
        d = int(np.ceil(np.log10(self._modulus)))
        fmt = 'mod\n{{:{:d}d}}'.format(d+1)
        if self._is_prime:
            fmt += '   (prime)'
        fmt += '   ({})'.format(str(self._residue.dtype))
        return print_to_string(self._residue) + fmt.format(self._modulus)

    def tile(self,s):
        r = np.tile(self._residue,s)
        return modular_residue(r, self._modulus, self._is_prime)

    def shape(self):
        return self._residue.shape
```

```
    def len(self):
        return len(self._residue)

    def reshape(self,*args):
        r = self._residue.copy()
        return modular_residue(r.reshape(*args), self._modulus, self._is_prime)

    def hstack(*args):
        return modular_residue(np.hstack(tuple([arg._residue for arg in args])), args[0]._modulus, args[0]._is_prim

    def vstack(*args):
        return modular_residue(np.vstack(tuple([arg._residue for arg in args])), args[0]._modulus, args[0]._is_prim

    def argmax(self,*args):
        return np.argmax(self._residue,*args)

    def min(self,*args):
        if args:
            return modular_residue(np.min(self._residue,*args),self._modulus,self._is_prime)
        else:
            return np.min(self._residue)

    def max(self,*args):
        if args:
            return modular_residue(np.max(self._residue,*args),self._modulus,self._is_prime)
        else:
            return np.max(self._residue)

    def transpose(self):
        return modular_residue(np.transpose(self._residue),self._modulus,self._is_prime)

    def astype(self,dtype):
        return modular_residue(self._residue.astype(dtype),self._modulus,self._is_prime)

    def enumerate(self,index=0):
        return enumerate(self._residue,index)

def print_to_string(*args, **kwargs):
    output = io.StringIO()
    print(*args, file=output, **kwargs)
    contents = output.getvalue()
    output.close()
    return contents



# Spaces.
def truth_space(S,M,dtype='int64'):
    # Every possible arange of S samples in modulus M, for universality and tightness tests (complete output space
    y = np.zeros((0,1))
    for i in range(S):
        y = np.vstack((np.tile(np.arange(0,M).reshape(-1,1),(1,M**i)).reshape(-1),np.tile(y,(1,M))))
    return modular_residue(y.astype(dtype),M)

def polynomial_space(x,E,S):
    # Power of modular residue samples at given exponents, for single input training.
    P = modular_residue(np.tile(x.residue().reshape(-1,1),(1,S)),x.modulus(),x.is_prime())
    if isinstance(E,modular_residue):
        E = E.residue()
    for i, e in enumerate(E):
        P[:,i] **= e
    return P
```

```
# Training.
def mles_solve(A,b):
   # Solve modular linear equation system Ax = b using Gaussian elimination.

   # Error check.
   if not isinstance(A,modular_residue) or not isinstance(b,modular_residue):
      raise TypeError('Tensors must be modular residues')
   if len(A.shape()) != 2 or len(b.shape()) != 2:
      raise ValueError('Tensors in linear system must be matrices')
   if A.shape()[0] != b.shape()[0]:
      raise ValueError('Matrices must have the same number of rows')

   # Cardinalities.
   M,N = A.shape() # number of rows and inputs
   if M > N:
      raise ValueError('Over determined system of equations')
   M,O = b.shape() # number of rows and outputs

   # Gauss elimination algorithm.
   T = A.hstack(b) # tableau
   p = np.arange(0,M) # pivot rows
   for k in range(M):
      i = k + T[p[k:],k].argmax() # pivot row
      p[k], p[i] = p[i], p[k] # swap pivot to current iteratino position
      T[p[k],k:] /= T[p[k],k:k+1] # normalize pivot row
      T[p[k+1:],k:] -= T[p[k]::M,k:]*T[p[k+1:],k:k+1] # elimination

   # Backward substitution.
   T = T[p,:] # sort tableau
   x = T[:,N:] if M==N else T[:,N:].vstack(modular_residue(np.zeros(N-M,O),A.modulus(),A.is_prime(),dtype=A
   for k in range(0,M-1):
      x[0:M-k-1,:] -= T[0:M-k-1,M-k-1:M-k]*x[M-k-1:M-k,:]
   return x

def fit(xh,yh,M=None,X=None,Eh=None,dtype=None,window=None):
   ''' Fit input to output using modular artificial intelligence model.

   Args:
      xh (numpy array or moduluar_residue): input samples
      yh (numpy array or moduluar_residue): output samples
      M (int, optional): specify modulus
      X (tuple of int, optional): specify input range
      Eh (tuple of numpy array or modular_residue): specify probing exponents
      dtype (str, optional): specify signed integer data type

   Retrun:
      (dict): modular artificial intelligence with fields:
         dtype (str): data type
         exponents (moduluar_residue): probing exponents for training space
         input (dict): input configuration with fields:
            encoder (function): input encoder
            minimum (numpy array): input minimum values
            maximum (numpy array): input minimum values
            size (tuple of int): input size
         model (string): AI model
         modulus (int): modulus
         output (dict): output configuration with fields:
            decoder (function): output decoder
            minimum (numpy array): output minimum values
            maximum (numpy array): output minimum values
```

```
        size (tuple of int): output size
      parameters (modular_residue): parameters for inference
      samples (int): number of samples
      spacer (function): create space for training
      time (float): elapsed training time [s]
'''

# Error check.
if not isinstance(xh,(np.ndarray,modular_residue)):
   raise TypeError('Input must be a numpy array or a modular residue')
if not isinstance(yh,(np.ndarray,modular_residue)):
   raise TypeError('Output must be a numpy array or a modular residue')
if Eh is not None and not isinstance(Eh,(np.ndarray,modular_residue)):
   raise ValueError('Probing exponents must be in a tuple of either numpy arrays or modular residues')
sx = xh.shape if isinstance(xh,np.ndarray) else xh.shape()
if not sx:
   raise ValueError('Empty input data not suppoted')
sy = yh.shape if isinstance(yh,np.ndarray) else yh.shape()
if not sy:
   raise ValueError('Empty output data not suppoted')
if sx[0] != sy[0]:
   raise ValueError('Inconsistent number of input and output samples')
if M is not None and not isinstance(M,int):
   raise TypeError('Modulus must be an integer')
if M is not None and M < 1:
   raise ValueError('Modulus must be a positive integer')
if isinstance(xh,modular_residue) and isinstance(yh,modular_residue) and xh.modulus() != yh.modulus():
   raise ValueError('Input and output modulus must be the same')
if X is not None and (not isinstance(X,(tuple,list,np.ndarray)) or not all([isinstance(x,int) for x in X])):
   raise TypeError('Input range must be a tuple of integers')

# Cardinalities.
if len(sx) == 1 or sx[1]==1:
   # Scalar model.
   N = (1,) # number of input variables
   O = (1,) if len(sy)==1 else (sy[1],) # number of input variables
   xh = xh if len(sx)==2 else xh.reshape(-1,1) # canonical output form
   yh = yh if len(sy)==2 else yh.reshape(-1,1) # canonical output form
   S = sx[0] # number of samples
   model = 'MAI' # AI model
elif len(sx) == 2:
   # Vector model.
   S = sx[0] # number of samples
   N = (sx[1],) # number of input variables
   O = (1,) if len(sy)==1 else (sy[1],) # number of input variables
   yh = yh if len(sy)==2 else yh.reshape(-1,1) # canonical output form
   model = 'VMAI' # AI model
else:
   raise ValueError('Unsupport data')
if M is None:
   if isinstance(xh,modular_residue):
      M = xh.modulus()
      xmin = np.array([xh[:,n].min() for n in range(N[0])]) # minimum input value
      xmax = np.array([xh[:,n].max() for n in range(N[0])]) # maximum input value
      if isinstance(yh,modular_residue):
         ymin = np.array([yh[:,o].min() for o in range(O[0])]) # minimum output value
         ymax = np.array([yh[:,o].max() for o in range(O[0])]) # maximum output value
      else:
         ymin = np.array([np.min(yh[:,o]) for o in range(O[0])]) # minimum output value
         ymax = np.array([np.max(yh[:,o]) for o in range(O[0])]) # maximum output value
   elif isinstance(yh,modular_residue):
      M = yh.modulus()
      xmin = np.array([np.min(xh[:,n]) for n in range(N[0])]) # minimum input value
```

```
      xmax = np.array([np.max(xh[:,n]) for n in range(N[0])]) # maximum input value
      ymin = np.array([yh[:,o].min() for o in range(O[0])]) # minimum output value
      ymax = np.array([yh[:,o].max() for o in range(O[0])]) # maximum output value
   else:
      xmin = np.array([np.min(xh[:,n]) for n in range(N[0])]) # minimum input value
      xmax = np.array([np.max(xh[:,n]) for n in range(N[0])]) # maximum input value
      ymin = np.array([np.min(yh[:,o]) for o in range(O[0])]) # minimum output value
      ymax = np.array([np.max(yh[:,o]) for o in range(O[0])]) # maximum output value
      M = next_prime(max([np.prod(xmax+1),np.max(ymax+1)])) # modulus
elif not is_prime(M):
   raise ValueError('Modulus must be a prime number')
if S <= N[0]:
   raise ValueError('Not enough samples')
if S > M:
   raise ValueError('Too many samples')
if X is None:
   X = xmax + 1 # input range

# Type definition.
if not isinstance(xh,modular_residue):
   xh = modular_residue(xh,M,True)
if np.unique(xh.residue(),axis=0).shape[0] < S:
   raise ValueError('Input must be unique')
if not isinstance(yh,modular_residue):
   yh = modular_residue(yh,M,True)
if dtype is None:
   dtype = 'int64'
   if M < 2**7:
      dtype = 'int8'
   elif M < 2**15:
      dtype = 'int16'
   elif M < 2**31:
      dtype = 'int32'
elif dtype not in ['int8', 'int16', 'int32', 'int64']:
   raise ValueError('Data type must be a signed integer')
xh.astype(dtype)
xh %= M
yh.astype(dtype)
yh %= M
xmin.astype(dtype)
xmax.astype(dtype)
ymin.astype(dtype)
ymax.astype(dtype)
if not isinstance(X,np.ndarray):
   X = np.array(X,dtype=dtype)

# Polynomial powers.
if Eh is None:
   # Exponent definition.
   Eh = modular_residue(np.arange(S,dtype=dtype),M,True)
else:
   # Error check.
   E = Eh % M
   if E.shape[0] != S:
      raise ValueError('There must be a unique probing exponent for each sample')
   Eu = np.unique(E)
   if Eu.shape[0] != E.shape[0]:
      raise ValueError('Probing exponents must be unique in modulus')
   Eh = modular_residue(E.astype(dtype),M,True)

# Training.
t = time.perf_counter(); # starting training time
Cx = lambda x, M=M, N=N, X=X, dtype=dtype : encode(x,M,N,X,dtype) # vector input encoder
```

```
    Cy = lambda y : decode(y) # output decoder
    Px = lambda x, E=Eh, S=S : polynomial_space(x,E,S) # input polynomial
    P = Px(Cx(xh)) # polynomial coefficients
    w = mles_solve(P,yh) # training itself
    t = round(time.perf_counter() - t,6) # elapsed training time

    # Output arguments.
    return {'model': model, 'parameters': w, 'timing': t, 'modulus': M, 'samples': S, 'dtype': dtype,
        'spacer': Px, 'exponents': Eh,
        'input': {'encoder': Cx, 'minimum': xmin, 'maximum': xmax, 'size': N},
        'output': {'decoder': Cy, 'minimum': ymin, 'maximum': ymax, 'size': O}
    }

def encode(x,M,N,X,dtype):
    x = x.astype(dtype)
    x = x if isinstance(x,modular_residue) else modular_residue(x,M,True)
    if len(x.shape()) == 1:
        x = x.reshape(-1,1)
    return modular_residue(
        np.sum(x.residue()*(np.cumprod(np.append(1,X[0:-1])).reshape(1,-1)),axis=1),
        x.modulus(),
        x.is_prime()
    )

def decode(y):
    return y

def infer(x,model):
    ''' Modular artificial intelligence inference.

    Args:
        x (numpy array or modular_residue): input
        model (dict): MAI model

    Returns:
        (modular_residue): model output given the input
    '''

    # MAI inference.
    return model['output']['decoder'](model['spacer'](model['input']['encoder'](x))@model['parameters'])

def save_model(model,file=None):
    ''' Convert MAI model to JSON string.

    Args:
        form (dict): model dictionary
        file (str, optional): file name to save JSON string (defaut None)

    Returns:
        (str): JSON string for the model
    '''

    # Data type conversion.
    model = copy.deepcopy(model) # avoid changing model outside this function
    model['parameters'] = model['parameters'].residue().astype(int).tolist()
    model['exponents'] = model['exponents'].residue().tolist()
    model['input']['minimum'] = model['input']['minimum'].tolist()
    model['input']['maximum'] = model['input']['maximum'].tolist()
    model['output']['minimum'] = model['output']['minimum'].tolist()
    model['output']['maximum'] = model['output']['maximum'].tolist()
    del model['input']['encoder']
    del model['output']['decoder']
    del model['spacer']
```

```
    # Form dictionary o JSON string.
    json_string = json.dumps(model, indent=4, ensure_ascii=False)

    # Save model JSON to file.
    if file is not None:
        with open(file, 'w') as fid:
            json.dump(model, fid, indent=4, ensure_ascii=False)

    # Output arguments.
    return json_string

def load_model(file):
    ''' Load MAI model from file.

    Args:
        file (str): file name to load JSON string into dictionary

    Returns:
        (dict): model dictionary
    '''

    # Load model dictionary from JSON file.
    with open(file, 'r') as fid:
        # Load model from file.
        model = json.load(fid)

        # Data type conversion.
        dtype = model['dtype'] # data type
        M = model['modulus'] # modulus
        S = model['samples'] # number of samples
        model['parameters'] = modular_residue(
            np.array(model['parameters'],dtype=dtype),
            M,
            True
        )
        model['exponents'] = modular_residue(np.array(model['exponents'],dtype=dtype),M,True)
        E = model['exponents']
        for end in ['input', 'output']:
            for limit in ['minimum', 'maximum']:
                model[end][limit] = np.array(model[end][limit],dtype=dtype)
        xmin = model['input']['minimum']
        xmax = model['input']['maximum']
        ymin = model['output']['minimum']
        ymax = model['output']['maximum']
        model['input']['size'] = tuple(model['input']['size'])
        N = model['input']['size']
        model['output']['size'] = tuple(model['output']['size'])
        O = model['output']['size']
        model['spacer'] = lambda x, E=E, S=S : polynomial_space(x,E,S)
        model['input']['encoder'] = lambda x, M=M, N=N, X=xmax+1, dtype=dtype : encode(x,M,N,X,dtype)
        model['output']['decoder'] = lambda y : decode(y)

        # Output arguments.
        return model



# Prime numbers.
def is_prime(n):
# Is prime predicate.
    if n < 4:
        return True
```

```
    imax = int(np.ceil(np.sqrt(n)))
    for i in range(2,imax+1):
        if (n % i) == 0:
            return False
    return True

def next_prime(n):
# Find next prime given a starting number.
    while not is_prime(n):
        n += 1
    return n



# Unit tests.
def test_wrong_instance():
    try:
        modular_residue([1,2],2)
        assert False
    except:
        pass

def test_wrong_instance():
    try:
        for dtype in ['int8', 'int16', 'int32', 'int64']:
            modular_residue(np.array([1,2],dtype=dtype),2,True)
    except:
        assert False

def test_attr():
    a = modular_residue(np.array([0,1,2,3,4,5],dtype='int64'),5,True)
    assert np.all(a[1:3] == np.array([1,2],dtype='int64'))
    a[1::2] = 2
    assert np.all(a == np.array([0,2,2,2,4,2],dtype='int64'))

def test_sum():
    a = modular_residue(np.array([0,1,2,3,4,5],dtype='int64'),5,True)
    b = modular_residue(np.array([2,2,2,2,2,2],dtype='int64'),5,True)
    c = modular_residue(np.array([2,3,4,0,1,2],dtype='int64'),5,True)
    assert np.all(a + b == c)
    assert np.all(a + b._residue == c)
    assert np.all(a + 2 == c)
    assert np.all(2 + a == c)

def test_sub():
    a = modular_residue(np.array([0,1,2,3,4,5],dtype='int64'),5,True)
    b = modular_residue(np.array([2,2,2,2,2,2],dtype='int64'),5,True)
    c = modular_residue(np.array([3,4,0,1,2,3],dtype='int64'),5,True)
    assert np.all(a - b == c)
    assert np.all(-(b - a) == c)
    assert np.all(a - b._residue == c)
    assert np.all(a - 2 == c)
    assert np.all(-(2 - a) == c)

def test_mul():
    da = np.array([0,1,2,3],dtype='int64')
    a = modular_residue(np.array([2**32]*4,dtype='int64')+da,5,True)
    b = modular_residue(np.array([2**32,2**33,2**34,2**35],dtype='int64'),5,True)
    c = modular_residue(np.array([1,4,2,2],dtype='int64'),5,True)
    assert np.all(a*b == c)
    assert np.all(a*b._residue == c)
    assert np.all((2**32)*a == da+1)
    assert np.all(a*(2**32) == da+1)
```

```
def test_pow():
   a = modular_residue(np.array([0]+[3]*6,dtype='int64'),5,True)
   b = np.array([0,0,1,2,3,4,5],dtype='int64')
   c = modular_residue(np.array([1,1,3,4,2,1,3],dtype='int64'),5,True)
   assert np.all(a**b == c)
   assert np.all(a**5 == a)
   a = modular_residue(np.array([0]+[3]*6,dtype='int64'),5,False)
   assert np.all(a**b == c)
   assert np.all(a**5 == a)

def test_div():
   a = modular_residue(np.array([1,2,3,4,3],dtype='int64'),5,True)
   b = np.array([1,1,1,1,2],dtype='int64')
   c = np.array([1,3,2,4,4],dtype='int64')
   assert np.all(1/(1/a) == a)
   assert np.all((1/a)*b == c)

def test_prime_predicate():
   N = [4, 8, 11, 47, 31243, 31249]
   B = [False, False, True, True, False, True]
   for b,n in zip(B,N):
      assert is_prime(n) == b

def test_spaces():
   assert np.all(truth_space(2,2) == np.array([[0, 0, 1, 1], [0, 1, 0, 1]]))
   assert np.all(truth_space(2,3) == np.array([[0, 0, 0, 1, 1, 1, 2, 2, 2], [0, 1, 2, 0, 1, 2, 0, 1, 2]]))
   x = modular_residue(np.array([0,1,2],dtype='int8'),3)
   assert np.all(polynomial_space(x,[0,1,2],3) == np.array([[1,0,0], [1,1,1], [1,2,1]]))

def test_mles_solve():
   dtype = 'int64'
   for M in [2, 3, 5]:
      P = polynomial_space(modular_residue(np.arange(M,dtype=dtype),M),np.arange(M),M)
      y = truth_space(M,M,dtype)
      w = mles_solve(P,y)
      assert np.all(P@w == y)

def test_mai():
   for M in [2, 3]: # modulus
      xh = np.arange(M) # input
      yh = truth_space(M,M) # all possible outputs
      model = fit(xh,yh) # MAI training
      assert np.all(infer(xh,model) == yh)

def test_vmai():
   N = 2 # number of input variables
   for X in [2, 3]: # range
      xh = truth_space(N,X).transpose() # input
      S = xh.shape()[0] # number of samples
      yh = truth_space(S,X) # all possible outputs
      M = next_prime(S)
      xh %= M # fix modulus
      yh %= M # fix modulus
      model = fit(xh,yh) # VMAI training
      assert np.all(infer(xh,model) == yh)

def test_cmai():
   N = 2 # number of input variables
   for X in [2, 3]: # range
      xh = truth_space(N,X).transpose() # input
      S = xh.shape()[0] # number of samples
      yh = truth_space(S,X) # all possible outputs
```

```
      M = next_prime(S)
      xh %= M # fix modulus
      yh %= M # fix modulus
      model = fit(xh,yh) # CMAI training
      assert np.all(infer(xh,model) == yh)

def unit_test():
   # List functions in unit test file.
   functions = [test_wrong_instance, test_wrong_instance, test_attr,
      test_sum, test_sub, test_mul, test_div, test_pow,
      test_prime_predicate, test_spaces, test_mles_solve,
      test_mai, test_vmai, test_cmai] # all unit test functions

   # Tests.
   format_str = '{{:{}s}} test {{}} of {{}}\r'.format(len(functions)+1) # formatting string
   pass_fail_str = '' # pass or fail string
   failures = [] # test failures
   for counter, function in enumerate(functions):
      # Perform test.
      try:
         function()
         pass_fail_str += '.'
      except Exception as e:
         pass_fail_str += 'F'
         failures.append(function)
      print(format_str.format(pass_fail_str, counter+1, len(functions)), end='', flush=True) # progress

   # Report.
   print(format_str.format(pass_fail_str, len(functions), len(functions)), flush=True)
   for fail in failures:
      print('fail at',fail.__name__)

   # Known specifications.
   print('\nKnown specifications:')
   print('1. modulus is a positive integer.')

# Test scripts.
if __name__ == "__main__":
   if len(sys.argv) > 1 and sys.argv[1].lower() == '-d':
      # Debug tests (temporary coded).
      if False:
         print(truth_space(2,2))
         print(truth_space(2,3))
         x = modular_residue(np.array([0,1,2],dtype='int8'),3)
         print(polynomial_space(x,[0,1,2],3))

      if True:
         M = 2 # modulus
         xh = np.arange(M) # input
         print('xh =',xh)
         yh = truth_space(M,M) # all possible outputs
         print('yh =',yh)
         model = fit(xh,yh) # MAI training
         print('y =',infer(xh,model))
         save_model(model,'mai.json')
         model = load_model('mai.json')
         print('y =',infer(xh,model))

      if True:
         X = 2 # input variable space
         N = 2 # number of input variables
         xh = truth_space(N,X).transpose() # input
         S = xh.shape()[0] # number of samples
```

```
        yh = truth_space(S,X) # all possible outputs
        M = next_prime(S)
        xh %= M # fix modulus
        yh %= M # fix modulus
        print('xh =',xh)
        print('yh =',yh)
        model = fit(xh,yh) # VMAI training
        print('w =',model['parameters'])
        print('y =',infer(xh,model))
        save_model(model,'vmai.json')
        model = load_model('vmai.json')
        print('y =',infer(xh,model))

   else:
      # Unit tests.
      unit_test()
```

The project configuration file is given by

```
[install]
use_pep517 = true

[build-system]
requires = ["setuptools >= 64"]
build-backend = "setuptools.build_meta"

[project]
name = "MAI"
version = "0.1"
requires-python = "==3.10.11"
dependencies = [
    "numpy==1.23.5"
]
```

A.1.2 Matlab

```
function [model,info] = mai_train(xh,yh,options)
%MAI_TRAIN Training of a modular artificial intelligence.
%   MODEL = MAI_TRAIN(X,Y) returns the modular artificial inteligence model
%   MODEL for the training set of input X and output Y pairs, comprised by
%   a struct with the fields:
%      decoder: output encoder function
%      exponents: polynomial space exponents for inference
%      encoder: input encoder function
%      error: model training error
%      infer: the inference function
%      inputs: input variable indexing for inference
%      parameters: the artificial intelligence trained parameters
%      refined: input refined variable indicator for inference
%
%   ... = MAI_TRAIN(X,Y,OPTIONS) allows to specify options OPTIONS for
%   training:
%      check: extra error check indicator {false, true} (default false)
%      group_size: input group size {2, 3, ...} (default 2)
%      modulus: fixed training modulus {1, 2, 3, 5, 7, 11, ...} (default 0)
%      refined: refinement indicator for training {false, true} (default true)
%      verbose: verbosity level {0, 1, 2...} (default 0)
%
%   [MODEL,INFO] = MAI_TRAIN(...) also returns a struct with training
%   information INFO containing the fields:
%      error: the training error at each layer
```

```
%      grouping: the group size used in the training
%      groups: the number of group at each layer
%      modulus: the modulus used in the modular algbra during training
%      parameters: the number of parameters at each layer
%      samples: the number of samples used in the training
%
%   Example:
%      M = 7; % modulus
%      xh = (0:M-1)'; % input training set
%      yh = randi(M,M,1)-1; % output training set
%      model = mai_train(xh,yh); % serial training
%      [model.infer(xh), yh] % parallel inference unit test
%
%   See also NEXT_PRIME, MODULAR_RESIDUE, POLYNOMIAL_SPACE.

%#ok<*AGROW>

% I hope my life worths a litte bit less than R$10,00: my dad is not easy
% :)

% criterion: hard coded

% input arguments
if nargin < 3, options = struct; end
if ~isfield(options,'verbose'), options.verbose = 0; end
if ~isfield(options,'refined'), options.refined = true; end
if ~isfield(options,'group_size'), options.group_size = 2; end
if ~isfield(options,'check'), options.check = false; end
if ~isfield(options,'modulus'), options.modulus = 0; end
if ~isfield(options,'input_step'), options.input_step = 0; end
if ~isfield(options,'best_error_reduction'), options.best_error_reduction = true; end

% parameters
S = size(xh,1); % number of samples
N = size(xh,2); % number of inputs
Ns = options.input_step; if ~Ns, Ns = N; end % input step
G = options.group_size; % grouping in each layer
M = next_prime(max([S;2^min(G,Ns);xh(:)+1;yh(:)+1]),1); % modulus
if options.modulus
   if options.modulus < M
      Mp = options.modulus; % fixed modulus
      rmax = ceil(log(max([xh(:);yh(:)])+1)/log(Mp)); % maximum refinement level
      M = Mp^rmax; % modulus
      refiners = {}; % input refiner function
      xhr = zeros(S,0); % refined input
      for r = 1:rmax
         refiners{r} = @(x)floor(rem(x*Mp^(r-1),M)/ceil(M/Mp));
         xhr = [xhr, refiners{r}(xh)]; % refine input
      end
      options.input_step = N; % lower refinement priority
      options.modulus = 0; % no modulus specification
      [model,info] = mai_train(xhr,yh,options);
      model.refiners = refiners;
      model.infer = @(x)model.decoder(...
         infer(model.encoder(x),...
         model.inputs,...
         model.refiners,...
         model.refined,...
         model.parameters,...
         model.exponents,...
         options.group_size));
      return
   else
```

```
      M = next_prime(options.modulus);
   end
end
lmax = M-1; % maximum refinement layer

% data
xlim = [min(xh(:)) max(xh(:))];
if any(rem(xh(:),1))
   encoder = @(x)floor((x-xlim(1))/(xlim(2)-xlim(1))/(1+eps)*M);
else
   encoder = @(x)x;
end
ylim = [min(yh(:)) max(yh(:))];
if any(rem(yh(:),1))
   yh = floor((yh-ylim(1))/(ylim(2)-ylim(1))/(1+eps)*M);
   decoder = @(y)y/(M-1)*(ylim(2)-ylim(1)) + ylim(1);
else
   decoder = @(y)y;
end
xh = modular_residue(encoder(xh),M,true); % input
yh = modular_residue(decoder(yh),M,true); % output

% training
Xh = {xh}; % input through layers
ey = yh; % current output error
info = struct('modulus',M,'samples',S,'grouping',G,...
   'inputs',N,'groups',0,'parameters',0,...
   'error',sum(ey.residue));
Ni = N; % number of current inputs
E = 0:S-1; % exponents
model = struct('parameters',modular_residue(zeros(S,0),M),...
   'inputs',{{num2cell(1:N)}},'refined',{{true(1,N)}},...
   'refiners',{{}},'infer',@(x)0,'exponents',E,...
   'encoder',encoder,'decoder',decoder,'xlim',xlim,'ylim',ylim);
while Ni
   % input groups
   gmax = min([Ns,Ni,G]); % maximum group number
   ng = nchoosek(Ni,gmax); % number of groups
   info.groups(1,end+1) = ng; % append number of groups

   % inputs
   if options.refined && N > 1 && numel(Xh) < lmax
      Xh(1,end+1) = {Xh{end}*G};
      model.inputs(1,end+1) = {num2cell(1:size(Xh{end-1},2))};
      model.refined(1,end+1) = {true(1,size(Xh{end-1},2))};
   else
      Xh(1,end+1) = {modular_residue(zeros(S,0),M)};
      model.inputs(1,end+1) = {{}};
      model.refined(1,end+1) = {[]};
   end

   % training iterations
   info.parameters(end+1) = 0;
   info.error(end+1) = sum(ey.residue);
   if options.verbose > 0, fprintf('\n%3d inputs starting with error %6d...\n',Ni,info.error(end)), end
   X = subsizes(M,gmax); % input fair subsizes: make sum(X) == M-1
   xs = scale(Xh{end-1}.residue,M,X); % scaled input
   for ig = 1:ng
      % choice for error reduction
      if options.best_error_reduction
         [g,dy,xhe,eyr] = best_reduction(xs,ey,gmax,M,X);
      else
         [g,dy,xhe,eyr] = search_non_null_error_reduction(xs,ey,gmax,M,X);
```

```
      end
      if ~dy, break, end

      % reduce error
      P = polynomial_space(xhe,E); % input polynomial space
      if options.verbose > 1, tic,fprintf('Solving a %d x %d modular system (L%02d G%03d-%03d)...',size(P,1),size
      w = P\eyr; % parameters
      yw = P*w; % next input
      if options.check && any(yw ~= eyr), error('Could not solve modular equation system'), end
      eyn = ey - yw; % update output error
      errorn = sum(eyn.residue); % new overal error
      if errorn < info.error(end)
         ey = eyn;
         Xh{end}(:,end+1) = yw;
         model.parameters(:,end+1) = w;
         model.inputs{end}(1,end+1) = {g};
         model.refined{end}(1,end+1) = false;
         info.error(end) = errorn;
         info.parameters(end) = info.parameters(end) + nnz(w);
      end
      if options.verbose > 1, fprintf(' error %6d in %6.3fs\n',info.error(end),toc), end
      if ~info.error(end), break, end
      if options.verbose > 0 && ~rem(ig,1e4), fprintf(' %6d of %6d groups analyzed\n',ig,ng), end
   end
   info.inputs(1,end+1) = size(Xh{end},2); % append number of inputs
   if ~info.error(end), break, end
   if Ni == 1, break, end
   Ni = size(Xh{end},2); % number of current inputs
end
model.infer = @(x)decoder(infer(...
   model.encoder(x),...
   model.inputs,...
   model.refiners,...
   model.refined,...
   model.parameters,...
   model.exponents,...
   options.group_size));
model.error = ey;
model.cascade = Xh;
end



function y = infer(x,i,r,t,w,E,G)
% cardinalities
nx = size(x,1); % number of inputs
nl = numel(i)-1; % number of layers
M = w.modulus; % modulus

% input refinement
if ~isempty(r)
   xr = r{1}(x);
   for ir = 2:numel(r)
      xr = [xr, r{ir}(x)];
   end
   x = xr;
end
n = size(x,2); % number o variables

% exponent pruning
ik = find(any(w.residue,2));
w = w(ik,:);
E = E(ik);
```

```
% inference
Xs = {modular_residue(x,M,true)}; % input
xw = modular_residue(zeros(nx,0),M,true); % inputs
y = modular_residue(zeros(nx,1),M,true); % output
iw = 0; % parameter counter
for il = 1:nl
   Xs{1,il+1} = Xs{il}*G;
   for ixw = 1:numel(t{il+1})
      if n == 1 || ~t{il+1}(ixw)
         X = subsizes(M,numel(i{il+1}{ixw}));
         xs = scale(Xs{il}.residue,M,X);
         xw(:,end+1) = group(xs(:,i{il+1}{ixw}),M,X);
         P = polynomial_space(modular_residue(round(xw.residue(:,end)),M),E);
         iw = iw + 1;
         y = y + P*w(:,iw);
         Xs{il+1}(:,end+1) = y;
      end
   end
end
y = y.residue;
end



function i = subs2ind(S,I)
if isscalar(S)
   o = cumprod([1,repmat(S,1,size(I,2))]); % offset
else
   o = cumprod([1,S]); % offset
end
for id = 2:size(I,2)
   I(:,id) = I(:,id).*o(id);
end
i = sum(I,2); % indexes
end



function Xi = subsizes(M,N)
Xi = floor(M^(1/N));
end



function x = scale(x,M,X)
if isscalar(X)
   x = floor(x/(M-1)/(1+eps)*X);
else
   x = floor(x/(M-1)/(1+eps)*diag(X));
end
end



function xg = group(x,M,X)
xg = modular_residue(subs2ind(X,x),M);
end



function ix = search_non_null_error_reduction(xh,yh,G)
iz = find(~yh);
```

```
for i = iz'
   for n = 1:size(xh,2)
      xh(xh(:,n)==xh(i,n),n) = 0;
   end
end
b = any(xh,1);
ix = find(b);
if G <= numel(ix)
   [~,is] = sort((xh(:,ix)>0)'*yh);
   ix = ix(is(end:-1:1));
   ix = ix(1:G);
elseif ~isempty(ix)
   nix = find(~b);
   ix = [ix nix(1:G-numel(ix))];
end
end



function [gb,dyb,xheb,yheb] = best_reduction(xh,yh,G,M,X)
g = nchoosek(1:size(xh,2),G);
dyb = 0;
gb = [];
xheb = [];
yheb = [];
for ig = 1:size(g,1)
   xhe = group(xh(:,g(ig,:)),M,X);
   [xhe,yhe] = prune_error(xhe,yh);
   dy = sum(yhe.residue);
   if dy > dyb
      xheb = xhe;
      yheb = yhe;
      dyb = dy;
      gb = g(ig,:);
   end
end
end



function [xh,yh] = prune_error(xh,yh)
[~,iu,ju] = unique(xh.residue); % unique encoded input
yh = accumarray(ju,yh.residue,[numel(iu),1],@(x)min(x)); % minimum error
yh = modular_residue(yh(ju),xh.modulus);
end



function p = next_prime(n,s)
if nargin < 2, s = 1; end
p = n;
while ~is_prime(p)
   p = p + s;
end
end



function b = is_prime(n)
b = true;
for d = 2:n^.5
   if ~rem(n,d)
      b = false;
```

```
      break
   end
end
end


function Px = polynomial_space(x,X)
if nargin < 2, X = 0:size(x,1)-1; end
Px = repmat(x,[1,numel(X)]);
for i = 1:numel(X)
   Px(:,i) = x.^X(i);
end
end


% criterion: long coded

classdef modular_residue
   %MODULAR_RESIDUE Number in modular algebra.
   %
   %   MODULAR_RESIDUE properties:
   %      isfinite - finite field indicator
   %      modulus - the modulus
   %      residue - the residue itself
   %
   %   MODULAR_RESIDUE methods:
   %      modular_residue - construct a number in modular algebra
   %
   %   Example: linear modular equations
   %      M = 2; % modulus
   %      X = modular_residue([0;1],M); % input
   %      Y = modular_residue([0,0,1,1;0,1,0,1],M); % output
   %      W = [ones(2,1) X] \ Y % universal weights and bias
   %
   %   See also INTERVAL, DOUBLE.

   properties
      %The residue itself.
      residue

      %The modulus.
      modulus

      %Finite field indicator.
      isfinite
   end

   methods
      % setters and getters
      function value = get.residue(r)
         value = r.residue;
      end

      function r = set.residue(r,value)
         % error check
         if ~isnumeric(value) || ~isreal(value)
            error('Residue must be real numbers.');
         end

         % setup
         r.residue = rem(value,r.modulus); %#ok<MCSUP>
```

```
   in = find(r.residue < 0);
   r.residue(in) = r.residue(in) + r.modulus; %#ok<MCSUP>
end

function value = get.modulus(r)
   value = r.modulus;
end

function r = set.modulus(r,value)
   % error check
   if ~isnumeric(value) || ~isreal(value) || ~isscalar(value)
      error('Modulus must be a positive integer.');
   end

   % setup
   r.modulus = value;
end

% constructor
function r = modular_residue(n,M,b)
   %MODULAR_RESIDUE Construct a number in modular algebra.
   %   R = MODULAR_RESIDUE(N,M) returns a residue in modular algebra
   %   for number N modulus M.
   %
   %   R = MODULAR_RESIDUE(N,M,B) allows to specify if the field is
   %   finite by the indicator B, which speeds up some operations.
   %
   %   See also MODULAR_RESIDUE.

   % input arguments
   if nargin < 3, b = false; end

   % setup
   r.modulus = M;
   r.residue = n;
   r.isfinite = b;
end

% residue operators
function r = plus(ra,rb)
   %PLUS Addition of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % minus operation
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      r = modular_residue(ra.residue+rb.residue,ra.modulus,ra.isfinite);
   elseif isa(ra,'modular_residue')
      r = modular_residue(ra.residue+rb,ra.modulus,ra.isfinite);
   else
      r = modular_residue(ra+rb.residue,rb.modulus,rb.isfinite);
   end
end

function r = minus(ra,rb)
```

```
   %MINUS Subtraction of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % minus operation
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      r = modular_residue(ra.residue-rb.residue,ra.modulus,ra.isfinite);
   elseif isa(ra,'modular_residue')
      r = modular_residue(ra.residue-rb,ra.modulus,ra.isfinite);
   else
      r = modular_residue(ra-rb.residue,rb.modulus,rb.isfinite);
   end
end

function r = uplus(r)
   %UPLUS Unary addition of residues.
   %
   %   See also MODULAR_RESIDUE.
end

function r = uminus(r)
   %UMINUS Unary subtraction of residues.
   %
   %   See also MODULAR_RESIDUE.

   % unary minus operation
   r = modular_residue(-r.residue,r.modulus,r.isfinite);
end

function r = times(ra,rb)
   %TIMES Multiplication of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % multiplication of residues
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      r = modular_residue(ra.residue.*rb.residue,ra.modulus,ra.isfinite);
   elseif isa(ra,'modular_residue')
      r = modular_residue(ra.residue.*rb,ra.modulus,ra.isfinite);
   else
      r = modular_residue(ra.*rb.residue,rb.modulus,rb.isfinite);
   end
end

function r = mtimes(ra,rb)
   %MTIMES Matrix multiplication of residues.
   %
```

```
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && size(ra,2)~=size(rb,1)
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % matrix multiplication of residues
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      r = modular_residue(ra.residue*rb.residue,ra.modulus,ra.isfinite);
   elseif isa(ra,'modular_residue')
      r = modular_residue(ra.residue*rb,ra.modulus,ra.isfinite);
   else
      r = modular_residue(ra*rb.residue,rb.modulus,rb.isfinite);
   end
end

function r = rdivide(ra,rb)
   %RDIVIDE Right division of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % right division of intervals
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      M = ra.modulus;
      b = ra.isfinite;
      ra = ra.residue;
      rb = rb.residue;
   elseif isa(ra,'modular_residue')
      M = ra.modulus;
      b = ra.isfinite;
      ra = ra.residue;
   else
      M = rb.modulus;
      b = rb.isfinite;
      rb = rb.residue;
   end

   % Euclidean algorithm
   rbi = [zeros(numel(rb),1), ones(numel(rb),1)];
   rmi = [repmat(M,numel(rb),1), rb(:)]; % Euclidean expansion of modulus
   in = find(rmi(:,2));
   while ~isempty(in)
      % Euclidean expansion
      q = floor(rmi(in,1)./rmi(in,2)); % quotient
      rbi(in,:) = [rbi(in,2), rbi(in,1)-q.*rbi(in,2)];
      rmi(in,:) = [rmi(in,2), rmi(in,1)-q.*rmi(in,2)];
      in = in(rmi(in,2)>0);
   end
   if ~all(rmi(:,1)==1 | rmi(:,1)==M)
      error('Divisor and modulus must be coprime.')
   end
```

```
   r = ra.*modular_residue(reshape(rbi(:,1),size(rb)),M,b);
end

function r = ldivide(ra,rb)
   %LDIVIDE Left division of residues.
   %
   %   See also MODULAR_RESIDUE.

   % symmetrical case
   r = rb ./ ra;
end

function r = mrdivide(ra,rb)
   %MRDIVIDE Matrix right division of residues.
   %
   %   See also MODULAR_RESIDUE.

   % matrix right division
   r = rb \ ra;
end

function r = mldivide(ra,rb)
   %MLDIVIDE Matrix left division of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~ismatrix(ra)
      error('Divisor must be a matrix.');
   end
   if ~ismatrix(rb)
      error('Dividend must be a matrix.');
   end
   if ~isscalar(ra) && ~isscalar(rb) && size(ra,1)~=size(rb,1)
      error('Inconsistent operand dimensions.')
   end
   if ~isscalar(ra) && isscalar(rb)
      error('Matrix dimensions must agree.');
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end
   if ~isa(ra,'modular_residue'), ra = modular_residue(ra,rb.modulus,rb.isfinite); end
   if ~isa(rb,'modular_residue'), rb = modular_residue(rb,ra.modulus,ra.isfinite); end

   % special cases
   if ra.modulus==1, r = modular_residue(zeros(size(ra,2),size(rb,2)),1,true); return, end
   if isscalar(ra), r = rb./ra; return, end

   % matrix left division
   na = size(ra,2); % number of columns in operator A
   nb = size(rb,2); % number of columns in operator B
   n = na + nb; % number of columns
   m = size(ra,1); % number of rows
   T = [ra,rb]; % tableau

   % Gaussian elimination
   s = 1:m; % row sorting vector
   for i = 1:m
      [~,imax] = max(subsref(T,substruct('()',{s(i:m),i}))); % pivot row
      s([i, i+imax-1]) = s([i+imax-1, i]); % swap pivot to current row
      Ts = subsref(T,substruct('()',{s(i),i+1:n}))/...
         subsref(T,substruct('()',{s(i),i})); % normalize pivot row
```

```
      T.residue(s(i),i+1:n) = Ts.residue;
      Ts = subsref(T,substruct('()',{s(i+1:m),i:n})) - ...
         subsref(T,substruct('()',{s(i+1:m),i}))*...
         subsref(T,substruct('()',{s(i),i:n})); % normalize ramaining rows
      T.residue(s(i+1:m),i:n) = Ts.residue;
   end

   % backward substitution
   for i = m-1:-1:1
      Ts = subsref(T,substruct('()',{s(i),na+1:n})) - ...
         subsref(T,substruct('()',{s(i),i+1:m}))*...
         subsref(T,substruct('()',{s(i+1:m),na+1:n}));
      T.residue(s(i),na+1:n) = Ts.residue;
   end
   r = [subsref(T,substruct('()',{s,na+1:n})); zeros(na-m,nb)];
end


function r = power(ra,rb)
   %POWER Exponentiation of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % power of residues
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      M = ra.modulus;
      b = ra.isfinite;
      a = ra.residue;
      e = rb.residue;
   elseif isa(ra,'modular_residue')
      M = ra.modulus;
      b = ra.isfinite;
      a = ra.residue;
      e = rb;
   else
      M = rb.modulus;
      b = rb.isfinite;
      a = rem(ra,M);
      in = find(a < 0);
      a(in) = a(in) + M;
      e = rb.residue;
   end
   if isscalar(a), a = repmat(a,size(e)); end
   if isscalar(e), e = repmat(e,size(a)); end
   r = ones(size(a));
   ar = a;
   er = e;
   if b
      r = rem(a.^rem(e,M-1),M);
   else
      while any(er > 0)
         iodd = find(rem(er,2));
         r(iodd) = rem(r(iodd).*ar(iodd),M);
         ar = rem(ar.*ar,M);
         er = floor(er/2);
```

```
      end
      r(~a & ~e) = 1;
   end
   r = modular_residue(r,M,b);
end

function B = lt(ra,rb)
   %LT Less than comparison of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % less than comparison of residues
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      B = ra.residue < rb.residue;
   elseif isa(ra,'modular_residue')
      B = ra.residue < rb;
   else
      B = ra < rb.residue;
   end
end

function B = gt(ra,rb)
   %GT Greater than comparison of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % greater than of residues
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      B = ra.residue > rb.residue;
   elseif isa(ra,'modular_residue')
      B = ra.residue > rb;
   else
      B = ra > rb.residue;
   end
end

function B = le(ra,rb)
   %LE Less than or equal comparison of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
```

```
   end

   % less or equal to comparison of residues
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      B = ra.residue <= rb.residue;
   elseif isa(ra,'modular_residue')
      B = ra.residue <= rb;
   else
      B = ra <= rb.residue;
   end
end

function B = ge(ra,rb)
   %GE Greater than or equal comparison of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % greater or equal to comparison of residues
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      B = ra.residue >= rb.residue;
   elseif isa(ra,'modular_residue')
      B = ra.residue >= rb;
   else
      B = ra >= rb.residue;
   end
end

function B = eq(ra,rb)
   %EQ Equality comparison of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % equal comparison of residues
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      B = ra.residue == rb.residue;
   elseif isa(ra,'modular_residue')
      B = ra.residue == rb;
   else
      B = ra == rb.residue;
   end
end

function B = ne(ra,rb)
   %NE Not equal comparison of residues.
   %
   %   See also MODULAR_RESIDUE.
```

```
   % error check
   if ~isscalar(ra) && ~isscalar(rb) && ~isequal(size(ra),size(rb))
      error('Inconsistent operand dimensions.')
   end
   if isa(ra,'modular_residue') && isa(rb,'modular_residue') && ra.modulus ~= rb.modulus
      error('Operand modulus must be the same.')
   end

   % not equal comparison of residues
   if isa(ra,'modular_residue') && isa(rb,'modular_residue')
      B = ra.residue ~= rb.residue;
   elseif isa(ra,'modular_residue')
      B = ra.residue ~= rb;
   else
      B = ra ~= rb.residue;
   end
end

function r = transpose(r)
   %TRANSPOSE Transpose of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~ismatrix(r), error('Residue must be a matrix.'), end

   % transpose operation
   r.residue = r.residue.';
end

function r = ctranspose(r)
   %CTRANSPOSE Complex conjugate transpose of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if ~ismatrix(r), error('Residue must be a matrix.'), end

   % transpose operation
   r.residue = r.residue';
end

function r = horzcat(varargin)
   %HORZCAT Horizontal concatenation of residues.
   %
   %   See also MODULAR_RESIDUE.

   % horizontal concatenation
   rs = cell(nargin,1);
   M = [];
   b = false;
   for i = 1:nargin
      if isa(varargin{i},'modular_residue')
         rs{i} = varargin{i}.residue;
         if isempty(M)
            M = varargin{i}.modulus;
            b = varargin{i}.isfinite;
         elseif M ~= varargin{i}.modulus
            error('Inconsistent modulus.')
         end
      elseif isnumeric(varargin{i})
         rs{i} = varargin{i};
      else
```

```
            error('Invalid value for concatenation.')
         end
      end
      r = modular_residue(horzcat(rs{:}),M,b);
   end

   function r = vertcat(varargin)
      %VERTCAT Vertical concatenation of residues.
      %
      %   See also MODULAR_RESIDUE.

      % vertical concatenation
      rs = cell(nargin,1);
      M = [];
      b = false;
      for i = 1:nargin
         if isa(varargin{i},'modular_residue')
            rs{i} = varargin{i}.residue;
            if isempty(M)
               M = varargin{i}.modulus;
               b = varargin{i}.isfinite;
            elseif M ~= varargin{i}.modulus
               error('Inconsistent modulus.')
            end
         elseif isnumeric(varargin{i})
            rs{i} = varargin{i};
         else
            error('Invalid value for concatenation.')
         end
      end
      r = modular_residue(vertcat(rs{:}),M,b);
   end

   function n = numArgumentsFromSubscript(varargin)
      n = 1;
   end

   function r = subsref(r,s)
      %SUBSREF Subscripted interval reference of residues.
      %
      %   See also MODULAR_RESIDUE.

      % subscripted residue reference
      for is = 1:length(s)
         if s(is).type == '.'
            r = r.(s(is).subs);
         else
            if isa(r,'modular_residue')
               r = modular_residue(subsref(r.residue,s(is)),r.modulus,r.isfinite);
            else
               r = subsref(r,s(is));
            end
         end
      end
   end

   function ind = end(r,k,~)
      ind = size(r.residue,k);
   end

   function r = subsasgn(r,s,rs)
      %SUBSREF Subscripted interval assignment of residues.
      %
```

```
   %   See also MODULAR_RESIDUE.

   % error check
   if ~isa(rs,'modular_residue') && ~isnumeric(rs)
      error('Invalid value for residues.')
   end
   if isa(rs,'modular_residue') && r.modulus ~= rs.modulus
      error('Modulus must agree.')
   end

   % subscripted residue assignment
   if (numel(s) == 1) && isequal(s(1).type, '()')
      if isa(rs,'modular_residue')
         r.residue(s.subs{:}) = rs.residue;
      else
         ri = rem(rs,r.modulus);
         in = find(ri < 0);
         ri(in) = ri(in) + r.modulus;
         r.residue(s.subs{:}) = ri;
      end
   elseif (numel(s) == 2) && isequal(s(1).type, '.')
      r.(s(1).subs)(s(2).subs{:}) = rs;
   end
end

% residue functions
function r = sum(r,varargin)
   %SUM Sum of residues.
   %
   %   See also MODULAR_RESIDUE.

   % sum of residues
   r.residue = rem(sum(r.residue,varargin{:}),r.modulus);
end

function r = cumsum(r,varargin)
   %CUMSUM Cumulative sum of residues.
   %
   %   See also MODULAR_RESIDUE.

   % cumulative sum of residues
   r.residue = rem(cumsum(r.residue,varargin{:}),r.modulus);
end

function r = reshape(r,varargin)
   %RESHAPE Resize of residues.
   %
   %   See also MODULAR_RESIDUE.

   % resize of residues
   r.residue = reshape(r.residue,varargin{:});
end

function varargout = size(r,varargin)
   %SIZE Size of residues.
   %
   %   See also MODULAR_RESIDUE.

   % size of residues
   varargout = cell(1,max(1,nargout));
   [varargout{:}] = size(r.residue,varargin{:});
end
```

```
function s = length(r)
   %LENGTH Length of residues.
   %
   %   See also MODULAR_RESIDUE.

   % length of residues
   s = length(r.residue);
end

function s = numel(r,varargin)
   %NUMEL Number of residues.
   %
   %   See also MODULAR_RESIDUE.

   % number of residues
   s = numel(r.residue,varargin{:});
end

function r = repmat(r,varargin)
   %REPMAT Repetition of residues.
   %
   %   See also MODULAR_RESIDUE.

   % repetition of residues
   r.residue = repmat(r.residue,varargin{:});
end

function b = isnan(r)
   %ISNAN Check whether residue is not a number.
   %
   %   See also MODULAR_RESIDUE.

   % not a number residue indicator
   b = isnan(r.residue);
end

function b = isinf(r)
   %ISINF Check whether residue is infinite.
   %
   %   See also MODULAR_RESIDUE.

   % infinite residue indicator
   b = isinf(r.residue);
end

function b = isempty(r)
   %ISEMPTY Check whether residue is empty.
   %
   %   See also MODULAR_RESIDUE.

   % empty residue indicator
   b = isempty(r.residue);
end

function b = isscalar(r)
   %ISSCALAR Check whether residue is scalar.
   %
   %   See also MODULAR_RESIDUE.

   % scalar residue indicator
   b = isscalar(r.residue);
end
```

```
function b = iscolumn(r)
   %ISCOLUMN Check whether residue is a column vector.
   %
   %   See also MODULAR_RESIDUE.

   % column residue indicator
   b = iscolumn(r.residue);
end

function b = isrow(r)
   %ISROW Check whether residue is a row vector.
   %
   %   See also MODULAR_RESIDUE.

   % row residue indicator
   b = isrow(r.residue);
end

function b = isvector(r)
   %ISVECTOR Check whether residue is a vector.
   %
   %   See also MODULAR_RESIDUE.

   % vector residue indicator
   b = isvector(r.residue);
end

function b = ismatrix(r)
   %ISMATRIX Check whether residue is a matrix.
   %
   %   See also MODULAR_RESIDUE.

   % matrix residue indicator
   b = ismatrix(r.residue);
end

function b = isnumeric(~)
   %ISNUMERIC Check whether residue is numeric.
   %   Always returns true.
   %
   %   See also MODULAR_RESIDUE.

   % numeric indicator
   b = true;
end

function [r,i] = min(r,varargin)
   %MINIMUM Minimum of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if (nargin >= 2) && ~isscalar(r) && ~isscalar(varargin{1}) && ...
         ~isequal(size(r),size(varargin{1}))
      error('Inconsistent operand dimensions.')
   end
   if isa(r,'modular_residue') && isa(varargin{1},'modular_residue') && ...
         r.modulus ~= varargin{1}.modulus
      error('Operand modulus must be the same.')
   end

   % minimum of residues
   if (nargin >= 2) && ~isempty(varargin{1})
```

```
      if isa(varargin{1},'modular_residue')
         [r.residue,i] = min(r.residue,varargin{1}.residue,varargin{2:end});
      elseif isnumeric(varargin{1})
         [r.residue,i] = min(r.residue,varargin{:});
      else
         error('Invalid second operand.')
      end
   else
      [r.residue,i] = min(r.residue,varargin{:});
   end
end

function [r,i] = max(r,varargin)
   %MAXIMUM Maximum of residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if (nargin >= 2) && ~isscalar(r) && ~isscalar(varargin{1}) && ...
         ~isequal(size(r),size(varargin{1}))
      error('Inconsistent operand dimensions.')
   end
   if (nargin >= 2) && isa(r,'modular_residue') && isa(varargin{1},'modular_residue') && ...
         r.modulus ~= varargin{1}.modulus
      error('Operand modulus must be the same.')
   end

   % maximum of residues
   if (nargin >= 2) && ~isempty(varargin{1})
      if isa(varargin{1},'modular_residue')
         [r.residue,i] = max(r.residue,varargin{1}.residue,varargin{2:end});
      elseif isnumeric(varargin{1})
         [r.residue,i] = max(r.residue,varargin{:});
      else
         error('Invalid second operand.')
      end
   else
      [r.residue,i] = max(r.residue,varargin{:});
   end
end

function r = cat(d,varargin)
   %CAT Concatenation of modular residues.
   %
   %   See also MODULAR_RESIDUE.

   % error check
   if nargin < 2, error('No enough input arguments.'), end

   % concatenation
   r = varargin{1}.residue;
   for i = 2:nargin-1
      if varargin{i}.modulus ~= varargin{1}.modulus
         error('Inconsistent modulus.')
      end
      r = cat(d,varargin{i}.residue);
   end
   r = modular_residue(r,varargin{1}.modulus);
end

function r = permute(r,dp)
   %PERMUTE Permute dimensions.
   %
```

```
        %   See also MODULAR_RESIDUE.

        % permutation
        r.residue = permute(r.residue,dp);
      end

      function r = abs(r)
        %ABS Absolute value.
        %
        %   See also MODULAR_RESIDUE.

        % absolute value
      end

      function r = floor(r)
        %FLOOR Floor value.
        %
        %   See also MODULAR_RESIDUE.

        % floor value
      end

      function r = ceil(r)
        %CEIL Ceil value.
        %
        %   See also MODULAR_RESIDUE.

        % ceil value
      end

      function r = round(r)
        %ROUND Rounded off value.
        %
        %   See also MODULAR_RESIDUE.

        % rounded off value
      end

      function n = nnz(r)
        %NNZ Number of non zeros.
        %
        %   See also MODULAR_RESIDUE.

        % non zeros count
        n = nnz(r.residue);
      end

      function disp(r)
        %DISP Display of residues.
        %
        %   See also MODULAR_RESIDUE.

        % display of residues
        disp(r.residue)
        fprintf('mod %d\n',r.modulus)
      end
    end
end
```

Index