Formal Verification of Constructions and Theorems on Hadamard Matrices

1052453

University of Oxford

A thesis submitted for the degree of

MSc in Mathematics and Foundations of Computer Science

Trinity Term, 2021

ABSTRACT

Hadamard matrices are constructions in linear algebra and combinatorics, which have broad applications in combinatorial designs, coding theory, sampling variability estimation, and etc. This project introduced the formalised definition of Hadamard matrices in the Lean theorem prover for inclusion in mathlib and implemented various constructions of Hadamard matrices, including the Sylvester's constructions and the Paley's constructions. This project also formalised other notions in algebra, such as the Kronecker product, the Hadamard product, quadratic residues, and the quadratic character, on the way to implementing Hadamard matrix constructions.
Keywords: interactive theorem proving, formal verification, Lean, Kronecker product, Hadamard product, quadratic residue, quadratic character, Hadamard matrix

ACKNOWLEDGEMENTS

I shall give deep thanks to my thesis supervisor Dr. Dmitrii Pasechnik, and deep thanks to my friend Yufeng Li, who has helped me with much LaTeX work. Many thanks also to people on the Lean Zulip chat group[1], who offered me various help, and Gabriel Moise, who provided me with an initial LaTeX template. Without the help from them, the completion of this project would have been impossible.

[1] https://leanprover.zulipchat.com

Contents

1 INTRODUCTION

1.1 Motivations

In discrete mathematics literature, there is an enormous amount of information on constructing, in one way or another, various sorts of "interesting" mathematical objects, e.g. linear and non-linear codes, Hadamard matrices, strong regular graphs, etc. There is considerable interest in having this information available in computer-ready form. However, usually the only available form is a paper describing the construction, while no computer code (and often no detailed description of a possible implementation) is provided.

Experience with implementing such constructions in SageMath[1] tells us that these constructions often contain bugs; some are easy to fix, while some might be crucial, as they invalidate theorems claimed, e.g. [6]. The most consequent remedy of such shortcomings is to implement formal verification of the pencil-and-paper constructions and proofs to eliminate possible bugs and gaps.

Such formal verification work is also meaningful from another perspective. Artificial intelligence (AI) has archived remarkable successes in a variety of domains. People are now wondering whether computers will be able to help humans with proofs, or whether an AI could come up with a proof of a theorem which has stumped humans. Such desires motivate the need of a encoded database of modern mathematics. Of course, real-world mathematics is rarely "fully encoded" into any foundational system, but experience shows that it is always possible in principle, and nowadays with computer proof assistants[2] it is becoming more common and feasible to do explicitly [13, p. 7].

For these reasons, communities surrounding proof assistants are passionately working on formalising the standard undergraduate-level mathematics curriculum, with the goal of building up to graduate and state-of-art levels results [10, p. 1].

This project provides formal verification of constructions in linear algebra and combinatorics such as Hadamard matrices, using proof assistant *Lean*[3] [8] and Lean's library *mathlib*[4] [12], which are under active development. Chapter 2 serves as an introduction to Lean.

1.2 Contributions

The main goal of this project is to formalise (in Lean) the definition of Hadamard matrices and implement constructions of Hadamard matrices. However, to achieve this aim, we need "scaffolds", such as the Kronecker product, which are/were not encoded in mathlib. As a result, this project implemented the following topics:

- the Hadamard product and basic results (Chapter 3)
- the Kronecker product and basic results (Chapter 3)
- quadratic residues, the quadratic character, and relevant results (Chapter 4)
- the definition of Hadamard matrices and basic results (Chapter 5)
- Sylvester's construction of Hadamard matrices (Chapter 6)
- Paley's constructions of Hadamard matrices (Chapter 7)
- a Hadamard matrix of order 92 (Chapter 8)
- the encoding of the Hadamard conjecture (sorry, not the proof!) (Chapter 9)
- some other core concepts in linear algebra (Chapter 3)

This project implements at least one Hadamard matrix of order n for every possible natural number $n \leqslant 112$, as well as for several infinite series of n, e.g. $n = 2^k$ (please refer to the conclusion chapter). All the source files created for this project are included in the Appendix.

[1] https://www.sagemath.org
[2] https://en.wikipedia.org/wiki/Proof_assistant
[3] https://leanprover-community.github.io
[4] https://leanprover-community.github.io/mathlib-overview.html

To my knowledge, the above topics covered in this project have never been introduced in mathlib before, except there is some ongoing parallel work on the Kronecker product, which so far has only concerned the definition and several rudimentary results. Therefore, the implementation of the topics above is all original, and this project will result in many additions to the matrix part and the combinatorics part of mathlib.

Some bits of this project has been merged into or are being merged into mathlib. The integration work will continue after the submission of this dissertation.

1.3 Notations and conventions

The thesis follows the following conventions (the concepts involved below will be introduced as the thesis goes on):

- the style `H i₁ i₂` is used for a fragment of the Lean code, and $H\ i_1\ i_2$ is used for a maths expression. Of course, sometimes the same expression can both be a maths expression and an expression appearing in the Lean code.
- `u`, `v`, ... denote universe variables.
- `α`, `β`, `γ`, ... denote generic types.
- `I`, `J`, `K`, ... denote finite types (in most parts).
- I or I_n may be used to denote the identity matrices when there is no confusion.
- $\mathbb{1}$ denotes the all-ones matrices, $\vec{1}$ denotes the column vectors/column matrices of ones.
- If H is a matrix, then $H\ i\ j$ or $H(i, j)$ denotes the (i, j)-th entry of H, and $H\ i$ denotes the i-th row or the row indexed by i of H, and $H\ \_\ j$ denotes the j-th column or the column indexed by j of H.
- F denotes a field (in fact, a finite field in most cases), p denotes the characteristic of F, and q denotes the cardinality of F when F is finite. F^* denotes the unit group of F.
- Other use of maths notations also follows the common conventions in general: for example, if H is a matrix, $H^\top$ is the matrix transpose of H.
- Throughout this thesis, all the matrices are finite dimensional.
- If `matrix.foo` is the identifier of the definition/construction of some matrix, which can be abbreviated as `foo` in the scope of `open matrix` or `namespace matrix`, then `Hadamard_matrix.foo` is the identifier of the proof that `foo` is a Hadamard matrix.
- In Lean, a "lemma" and a "theorem" are precisely the same thing, so we may use them interchangeably.
- Whenever a theorem is stated, the mathematical proof will resemble the Lean code solution.
- Some simple maths proofs or implemented proofs can be omitted in the main body of the thesis, for the purpose of focusing on the core thread of the project and not widely exceeding the recommended dissertation length. Please see the appendix for the complete implementation.
- *LHS* is an abbreviation of "left hand side", and *RHS* is an abbreviation of "right hand side".

The chapter/section-scope conventions will be introduced at the beginning of the corresponding chapter/section. The Lean source files follow the style conventions for writing mathlib in general. Please see https://leanprover-community.github.io/contribute/index.html for more information about the mathlib style conventions.

1.4 Other comments

This thesis is not aimed to serve as an introduction of how to program or write maths in Lean, nor an overview of the structure of the up-to-date mathlib. The main focus of this thesis is to illustrate the maths of the topics that will be covered and the Lean implementation of these topics. Therefore, one usually needs maths maturity and preliminary experience of using Lean as a theorem proved to fully understand both the maths background and the Lean code in this project[1].

[1] I had no experience at all with Lean before starting this project. I leaned Lean myself with the help from my supervisor and people on Zulip in the Trinity term.

Formalised proofs amplify subtleties of pencil-and-paper proofs and fill all the possible gaps a pencil-and-paper proof may have. The implementation I am going to exhibit may seem straightforward and not that "tedious", because I aim to finally obtain a version of implementation that is concise, general, and compatible with other parts of mathlib, after many "bad" attempts, and also because some technical subtleties or ugliness are hidden on purpose (see the linked remark on p. 22 for an example) to focus on the main subject.

2 THE LEAN THEOREM PROVER

Formal verification, including the verification of ordinary mathematical theorems, involves the use of logical and computational methods to establish claims that are expressed in precise mathematical terms [2, p. 1]. The Lean project was launched by Leonardo de Moura at Microsoft Research Redmond in 2013 [1, 2], and is currently an open-source project. Lean itself is a programming language, but is often used, and will be used in this project, as an interactive theorem prover/proof assistant. Unlike proof assistants based on ZFC[1] axiomatics, the formal system of Lean is a dependent type theory based on the calculus of inductive constructions [7, 9], which is as well powerful enough to prove almost any conventional mathematical theorem, and expressive enough to do it in a natural way [2, p. 2]. Other well-known type theory based proof assistants are Coq [15][2] and Agda [14][3]

Books [1, 2] are usually recommended for Lean beginners. This section characterises several important features of Lean, with particular attention to features involved in the following chapters. This section is not aimed to be a comprehensive introduction to Lean for Lean learners as giving a thorough introduction in a short chapter just not seems possible. In fact, even after studying [1, 2], one usually still needs to consult experts, e.g. on Zulip, in the course of work.

2.1 Dependent type theory

In a nutshell, Lean is a tool for building complex expressions in the language of dependent type theory [1, p. 1]. "Type theory" gets its name from the fact that every expression has an associated type [2, p. 5]. One can use the *#check* command to print the type of an expression in Lean. Here are several examples:

```
#check 2 -- prints `2 : ℕ` as Lean view `2` as a natural number by default
#check (2 : ℚ) -- prints `2 : ℚ` if we claim the given `2` is a rational number
#check 2 + 3  -- prints `2 + 3 : ℕ`

def f (x : ℕ) := 2 * x  -- this defines a function
#check f  -- prints `f : ℕ → ℕ`

#check (4, 5) -- prints `(4, 5) : ℕ × ℕ`,
              -- where `×` is the notation of the operation `prod`

#check ℕ -- prints `ℕ : Type`
#check ℕ × ℕ -- prints `ℕ × ℕ : Type`
```

`ℕ`, `ℤ`, `ℚ`, are syntax sugars for `nat`, `int`, `rat`, respectively. All the mathematical statements/propositions have type `Prop`. If `p` has type `Prop`, an expression has type `p` if and only if it is a proof of proposition `p`.

```
def p := 2 + 3 = 5 -- defines `p` to be the statement `2 + 3 = 5`
#check p  -- prints `p : Prop`

lemma pf : p := by simp [p] -- this constructs `pf1` as a proof of `p`
#check pf  -- prints `pf : p`
```

The type `Prop` is proof-irrelevant, in the sense that any two proofs of the same proposition are equal. The lemma below shows that any two arbitrary proofs `pf1` and `pf2` of a given proposition `P` are equal.

```
lemma irrelevant (P : Prop) (pf1 pf2 : P) : pf1 = pf2 := by simp
```

[1] Zermelo-Fraenkel + axiom of Choice
[2] see also https://en.wikipedia.org/wiki/Coq
[3] see also https://en.wikipedia.org/wiki/Agda_(programming_language)

We have known that expressions like `ℕ`, `ℕ × ℚ`, `bool` have type `Type`, it is natural to ask what type dose `Type` itself have as every expression in Lean has a type. It has type `Type 1`, and `Type 1` has `Type 2`, and so on.

```
#check Type -- Type : Type 1
#check Type 1 -- Type 1 : Type 2
```

In fact, Lean has a countably infinite noncumulative hierarchy of types with `Prop` at the bottom: `Prop`: `Type`: `Type 1`: `Type 2`: `...`. Every type in this hierarchy is called a universe. `Type` is an abbreviation of `Type 0` and contains common types such as `Prop`, `ℕ`, `ℤ`, `ℚ`, `ℝ`, `ℕ → ℕ`, `...`. An arbitrary `Type u`, where `u` is called a universe variable, can also be expressed as `Type*` to avoid giving the arbitrary universe a name. The following is a way to declaring universe variables explicitly:

```
universe u -- declares `u` to be a universe variable
variable a : Type u -- declares `a` to be a variable in `u`
#check a  -- prints `a : Type u`
```

The variable `a` defined in the above example is called a (universe) polymorphic variable as it lies in an arbitrary universe.

There are also polymorphic functions, such as `list` and `prod` in the following examples, in the sense that at least one parameter of them can lie in different universes. Such polymorphic functions explain what makes dependent type theory dependent: types can depend on parameters [2, p. 17].

```
variables α : Type* -- declares variables `α`
variable β : Type 2 -- declares variables `b`

#check α -- prints `α : Type u_1`
#check β -- prints `β : Type 2`

#check list  -- prints `list : Type u_1 → Type u_1`,
             -- where `u_1` is a variable ranging over type levels
#check list ℕ  -- prints `list ℕ : Type`
#check list α -- prints `list α : Type u_1`

#check prod -- prints `prod : Type u_3 → Type u_4 → Type (max u_3 u_4)`
#check ℕ × ℕ -- prints `ℕ × ℕ : Type`
#check ℕ × β -- prints `ℕ × β : Type 2`
#check α × β -- prints `α × β : Type (max u_1 2)`
```

Who is more comfortable with set-theoretic foundations can analogously think of a type as nothing more than a set, in which case, the elements of the type are just the elements of the set [2, p. 7]. Indeed, as the implementations in this project do not involve many set/type-axiomatic foundations, we may not distinguish a set and a type in proofs in words when it is safe to do so.

2.2 Namespaces

Lean provides the ability to group declarations into nested, hierarchical namespaces, see [2, p. 15]. When we declare that we work in the namespace `foo`, every identifier we declare has a full name prefixed by "foo". Within `foo`, we can refer to identifiers by their shorter names omitting the prefix, but once we end `foo`, we have to use the longer names, unless we later reopen `foo`.

```
namespace foo -- declares we are working in the namespace `foo`

-- Now, we are working in the namespace `foo`
```

```
def b : ℕ := 10
def f (x : ℕ) : ℕ := x + 5
def fb : ℕ := f b

namespace bar -- declares we are working in the namespace `bar`

-- Now, we are working in the namespace `foo.bar`

def ffb : ℕ := f (f b)

#check fb
#check ffb

end bar

-- Now, we back to the namespace `foo`

#check fb
-- `#check ffb` will reports error: unknown identifier 'ffb'.
#check bar.ffb

end foo

#check foo.fb
#check foo.bar.ffb

namespace foo -- We can redo `namespace foo`.
def g (x : ℕ) : ℕ := b + 5
#check g
end foo

-- #check g -- error
#check foo.g

open foo -- `open foo` allows us to use the shorter name `name`
         -- for any identifier in form `foo.name`,
         -- but if we declare an identifier `g` after `open foo`,
         -- `g`'s full name is `g`, not `foo.g`.

#check g
#check bar.ffb
```

2.3 Tactics

Lean offers two approaches to construct a proof: (1) writing proof terms, and (2) using tactics. A proof term is a direct representation of a mathematical proof, or in other words, is the expression (a proof is an expression) itself; while tactics are commands, or instructions, that describe how to construct such a proof/expression [2, p. 53]. We describe proofs that consist of sequences of tactics as "tactic-style" proofs and the ways of writing proof terms as "term-style" proofs [2, p. 53]. "Tactic-style" proofs are more often used for complex proofs. I am not going to introduce every single tactic which appears in this thesis. [17] documents all the commonly used tactics.

The code below shows an example of a "term-style" proof and an example of "tactic-style" proof. `theorem` or `lemma` creates a goal of constructing a term with expected type. In the examples below, `p ∧ q ∧ p` is the expected type, and `p q : Prop`, `hp : p`, and `hq : q` are hypotheses. In words, we have `p q` as propositions, and `hp` is a proof of `p`, and `hq` is a proof of `q`.

```
-- a term-style proof
lemma pf1 (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p :=
and.intro hp (and.intro hq hp)

#check and.intro -- prints `and.intro : ?M_1 → ?M_2 → ?M_1 ∧ ?M_2`,
                 -- where `?M_1` and `?M_2` denotes indeterminate variables

-- a tactic-style proof
lemma pf2 (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p :=
begin -- the `begin ... end` block allows us to write a list of tactics
  split, -- splits the goal `p ∧ q ∧ p` into two goals `p` and `q ∧ p`
  -- the first goal is `p`
  -- I.e. Lean expects us to provide an expression of type `p`
  exact hp, -- provides an exact proof term `hp`
            -- this closes the first goal as `hp` has type `p`
  split, -- splits `q ∧ p` into `q` and `p`
  exact hq, -- provides an exact proof term `hq`
  exact hp, -- provides an exact proof term `hp`
end
```

`split` and `exact` above are both tactics. The `begin ... end` block allows us to write a list of tactics, separated by commas. Equivalently, we can use `by {...}`.

Term-style proofs and tactic-style proofs are not mutually exclusive. Indeed, `hp` and `hq` are provided as proof terms after `exact` in `pf2`. `pf3` below is also an example of a mixture.

```
-- mix
lemma pf3 (p q : Prop) (hp : p) (hq : q) : p ∧ q ∧ p :=
begin
  split,
  exact hp, -- `hp` is a proof term
  exact and.intro hq hp -- `and.intro hq hp` is a proof term
end
```

2.4 Structures

A structure or a record in Lean is a type which contains at least one or usually servery fields. We have seen an example of structures: the product type, `prod`, is a structure containing two fields.

```
-- the following is the definition of `prod` in mathlib
structure prod (α : Type u) (β : Type v) :=
(fst : α) -- the name of the first field is given to be `fst`
(snd : β) -- the name of the second field is given to be `snd`
#check prod -- prod : Type u_1 → Type u_2 → Type (max u_1 u_2)

variables α β : Type*
#check prod α β -- α × β : Type (max u_1 u_2)
                -- α × β is a new type and itself has type Type (max u_1 u_2)
```

We can construct an instance of `α × β` by the function `prod.mk`, which is automatically generated when defining `prod` using the command `structure`, or by using the *anonymous constructor*, `⟨arg1, arg2, ...⟩`, if the type of the constructed expression is clear. Also, `(arg1, arg2)` is a specially assigned notation for constructing instances of product structures.

```
variables (a : α) (b : β)

#check prod.mk a b -- (a, b) : α × β
#check (⟨a, b⟩ : prod α β) -- (a, b) : α × β
#check (a, b) -- (a, b) : α × β
```

Meanwhile, when the `structure` command defines a structure `S`, Lean also generates projection functions that allow us to "destruct" each instance of `S` and retrieve the values that are stored in its field [2, p. 135].

```
#check prod.fst -- prod.fst : ?M_1 × ?M_2 → ?M_1
#reduce prod.fst (10, 20) -- 10
#reduce (10, 20).1 -- 10

#check prod.snd -- prod.snd : ?M_1 × ?M_2 → ?M_2
#reduce prod.snd (10, 20) -- 20
#reduce (10, 20).2 -- 20
```

Of course, we can define new structures we like. For example:

```
structure my_str := -- a new structure with thtree fields
(f1 : ℕ) (f2 : ℕ) (f3 : ℕ)

variables n : ℕ
-- defines `triple n` to be `⟨n, n + n, n + n + n⟩`of type `my_str`
def triple : my_str := ⟨n, n + n, n + n + n⟩

#check triple n -- triple n : my_str
#reduce (triple n).1 -- n
#reduce (triple n).2 -- n.add n
#reduce (triple n).3 -- (n.add n).add n
#reduce my_str.f3 (triple 5) -- 15
#reduce (triple 5).3 -- 15
```

2.5 Type classes

Any family of types can be marked as a *type class*, and we can then declare particular elements of a type class to be *instances* [2, p. 141]. The declaration of instances provides hints to *type class inference*, a mechanism for Lean's elaborator: any time the elaborator is looking for an element of a type class, it can consult a table of declared instances to find a suitable element.

Let's illustrate type classes by a simple example. `has_add` is a type class in mathlib defined as follows:

```
class has_add (α : Type u) := (add : α → α → α)
```

The above definition is a shorthand for

```
@[class] structure has_add (α : Type u) := (add : α → α → α)
```

I.e. we mark the structure family `has_add` as a type class by adding the tag `@[class]`.

`@[...]` or `attribute [...]` is the way to tag *attributes*. Attributes are a tool for associating information with declarations [16]. Please see [16] for more information about attributes.

An element of the class `has_add α` is simply an expression of the form `has_add.mk f`, for some element `f : α → α → α`. We can declare an element of `has_add α` as an instance of `has_add α` by the command `instance` so that it can then be used in type class inference. In words, `has_add α` has an element/instance means `α` has a binary operation called "addition", denoted as `+`. For example, `nat.has_add` is an instance defined in mathlib of type `has_add nat`.

We can construct an instance of `has_add nat` ourselves:

```
-- We construt an element `my_inst` of `has_add ℕ`.
def my_inst : has_add ℕ :=
⟨nat.add⟩ -- `nat.add` is a function of type ℕ → ℕ → ℕ defined in mathlib
-- Next, we declare `my_inst` to be an instance of `has_add ℕ`.
attribute [instance] my_inst
```

Equivalently, we can do

```
@[instance] def my_inst : has_add ℕ := ⟨nat.add⟩
```

or

```
instance my_inst : has_add ℕ := ⟨nat.add⟩
```

We now illustrate how type class inference works by a simple example. As `nat` has addition, we can define addition on `nat × nat` to be the point-wise addition:

```
instance my_inst' : has_add (ℕ × ℕ):= ⟨λ ⟨a, b⟩ ⟨c, d⟩, ⟨a + c, b + c⟩⟩
```

The addition `+` over `nat` makes sense, as the elaborator can find an instance of `has_add nat`. In contrast, if we change `nat` to an arbitrary type `α` without a built instance of `has_add α`, the code

```
instance my_inst'' : has_add (α × α):= ⟨λ ⟨a, b⟩ ⟨c, d⟩, ⟨a + c, b + c⟩⟩
```

will report the error "failed to synthesise type class instance for `has_add α`".

Lean provides a form of *inheritance* between structures/classes: we can *extend* existing structures by adding new fields [2, p. 139].

```
class has_one (α : Type u) := (one : α)

class has_mul (α : Type u) := (mul : α → α → α)

class semigroup (G : Type u) extends has_mul G :=
(mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))

class mul_one_class (M : Type u) extends has_one M, has_mul M :=
(one_mul : ∀ (a : M), 1 * a = a)
(mul_one : ∀ (a : M), a * 1 = a)

class monoid (M : Type u) extends semigroup M, mul_one_class M :=
(npow : ℕ → M → M := npow_rec)
(npow_zero' : ∀ x, npow 0 x = 1 . try_refl_tac)
(npow_succ' : ∀ (n : ℕ) x, npow n.succ x = x * npow n x . try_refl_tac)
```

This feature enables us to build algebraic hierarchies and define algebraic structures (the above is an example) in a convenient way.

Type classes as well as type class inference is a more involved topic. Please refer to Chapter 10 of [2] for more information about type classes, and refer to Section 6.10 of [2] for more information about Lean's elaborator.

2.6 Coercions

In Lean, the type of natural numbers, `nat` (also denoted as `ℕ`), is not a "subset/subtype" of the type of integers, `int` (also denoted as `ℤ`). Namely, a natural number in Lean is not by definition, or definitionally equal to, an integer. However, there is a function `int.of_nat` in mathlib that embeds the natural numbers into the integers such that we can view a natural number as an integer when needed [2, p. 89]. Such a function is called a *coercion*.

Coercions are transitive in the sense that if we have a built coercion from natural numbers to integers and a built coercion from integers to rationals (indeed, we have `rat.of_int`), then Lean produces a coercion from natural numbers to rationals automatically when needed.

Lean also has mechanisms to automatically detect and insert coercions [2, p. 89].

```
variables m n : ℕ
variables i : ℤ
variables j : ℚ
#check i + m -- i + ↑m : ℤ
#check i + m + n -- i + ↑m + ↑n : Z
#check j + i + m -- j + ↑i + ↑m : ℚ
```

The output of the *#check* command show that a coercion has been automatically inserted by printing an arrow.

However, such mechanisms to insert necessary coercions are not almighty. Sometimes, we have to insert coercions manually. In the examples below, when the order of arguments changes, Lean is not able to recognise the need for a coercion until it has already parsed the earlier arguments [2, p. 89].

```
-- #check m + i -- error
#check ↑m + i -- ↑m + i : ℤ
#check ↑(m + n) + i -- ↑(m + n) + i : ℤ
#check ↑m + ↑n + i -- ↑m + ↑n + i : ℤ
#check ↑i + ↑m + j -- ↑i + ↑m + j : ℚ
```

Coercions allows us to view an element of one type `α` as an element of a "bigger" or "equivalent" type `β`. In Lean, coercions are implemented on top of the type class resolution framework [2, p. 149]. We can build a coercion from `α` to `β` by declaring an instance of `has_coe α β`. For example, we can define a coercion from real numbers to complex numbers as follows:

```
instance my_coe : has_coe ℝ ℂ := ⟨λ r, ⟨r, 0⟩⟩
variables (r : ℝ) (c : ℂ)
#check c + r -- c + ↑r : ℂ
```

2.7 Lean's simplifier

This section is excerpted and rephrased from [18], which is a document about Lean's simplifier.

Lean has a "simplifier", called `simp`, which itself is a tactic. Lean's simplifier is what is called a "conditional term rewriting system". Rewriting is the act of changing a goal of the form `P(A)` into `P(B)`, given a hypothesis of the form `A = B` or `A ↔ B`. The simplifier's job is to try and prove, or at least simplify, goals or hypotheses, by using repeated rewrites.

`@[simp]` can be used to tag suitable lemmas with the `simp` attribute. We call lemmas tagged in this way "simp lemmas". Here are some examples of simp lemmas in mathlib:

```
@[simp] theorem nat.dvd_one {n : ℕ} : n ∣ 1 ↔ n = 1
@[simp] theorem mul_eq_zero {a b : ℕ} : a * b = 0 ↔ a = 0 ∨ b = 0
@[simp] theorem list.mem_singleton {a b : α} : a ∈ [b] ↔ a = b
@[simp] theorem set.set_of_false : {a : α | false} = ∅
```

When the simplifier is attempting to simplify a term `T`, it looks through the simp lemmas known to the system at that time, in a sensible way, and if it runs into a lemma of the form `A = B` or `A ↔ B`, for which `A` shows up in `T`, it rewrites the lemma at `T` to replace it with a `B`, and starts again.

Here is an example using `simp`:

```
import algebra.group.defs
variables (G : Type) [group G] (a b c : G)
example : a * a⁻¹ * 1 * b = b * c * c⁻¹ :=
by
  simp
end
```

Please see [18] for more information about the simplifier and the use and variants of the `simp` tactic.

3 KRONECKER PRODUCT AND HADAMARD PRODUCT

The Kronecker product, which is/was missing in mathlib, is widely used in the constructions of Hadamard matrices. As a preparation, this section discusses the implementation of the Kronecker product, the Hadamard product, and some other basic concepts about matrices.

- The implementation of the Kronecker product follows [22], and the implementation of the Hadamard product follows [21].
- Please see file MAIN1.LEAN for the complete code work of the Kronecker product and the Hadamard product.

3.1 Matrix

This section introduces some basic Lean lemmas concerning matrices.

In mathlib, the implementation of matrices is based on the concept "finite type", and the implementation of "finite type" is based on the concept "finite set" in turn. Mathlib encodes the two concepts as follows:

```
/-- `finset α` is the type of finite sets of elements of `α`. It is implemented
  as a multiset (a list up to permutation) which has no duplicate elements.
  `finset.nodup` is the poof that `finset.val` has no duplicate elements. -/
structure finset (α : Type*) :=
(val : multiset α)
(nodup : nodup val)

/-- `fintype α` means that `α` is finite, i.e. there are only
  finitely many distinct elements of type `α`. The evidence of this
  is a finset `elems` (a list up to permutation without duplicates),
  together with a proof `complete` that everything of type `α` is in the list. -/
class fintype (α : Type*) :=
(elems [] : finset α)
(complete : ∀ x : α, x ∈ elems)
```

`finset α` is implemented as a structure, while `fintype` is implemented as a class, because there can be many instances in `finset α` as finite sets with elements in `α`, while there is only one instance (up to equivalence) in `fintype α`, which proves that type `α` is finite; thus, implementing `fintype α` this way enables the elaborator to automatically search for the evidence that `α` is a finite type.

In mathlib, "matrix" is implemented as follows:

```
/-- `matrix I J` is the type of matrices whose rows are indexed by the fintype `I`
    and whose columns are indexed by the fintype `J`. -/
def matrix (I : Type u) (J : Type u') [fintype I] [fintype J] (α : Type v) :
Type (max u u' v) :=
I → J → α
```

In the list of parameters to `matrix`, round brackets mark parameters explicitly supplied by the user, such as `(I : Type u)`. Curly brackets are also commonly used in other cases, which mark implicit parameters inferred by Lean from the context. `[fintype I]` is an abbreviation of `[inst : fintype I]` with some arbitrary identifier `inst` of the instance omitted, and so is `[fintype J]`. Square brackets mark instance parameters inferred by the type class system. Thus `matrix I J α` is a type of matrices when Lean can infer that `I, J` are finite types. A matrix of type `matrix I J α` is defined to be a map of type `I → J → α`. For `M` in `I → J → α` (or equivalently in `matrix I J α`), `i` in `I`, and `j` in `J`, `M i j` is an element in `α` and called an entry of matrix `M`. We call `I` the row index type of `M`, and `J` the column index type of `M`.

Caution! As far as I know, while I am writing this thesis, some people in the Lean community are working removing the "finite type" requirement and generalising the definition to not only finite dimensional matrices. As such, this core definition may change in the coming month! For our purpose and in view of the current state of mathlib, all the matrices in this thesis will be finite dimensional.

In the linear algebra part of mathlib, a vector is literally a map from some finite type and is not implemented separately, and conversely, a map from some finite type is also viewed as a vector. In particular, `1` is a vector of ones supposed the type it maps to has an element called one.

```
variables (I α : Type*) [fintype I] [has_one α]
#check (1 : I → α)
```

The square brackets in the variable declaration enable the type class inference to make use of the instances.

In mathlib, the matrices which have a single column (also called column matrices) and the matrices which have a single row (also called row matrices) are implemented as follows:

```
/-- `matrix.col w` is the column matrix whose entries are given by vector `w`. -/
def col (w : I → α) : matrix J unit α
| x y := w x

/-- `matrix.row v` is the row matrix whose entries are given by vector `u`. -/
def row (v : I → α) : matrix unit I α
| x y := v y
```

In particular, `col 1` is the column matrix of ones, and `row 1` is the row matrix of ones.

In mathlib, the identity matrix of type `matrix I I α` is not defined separately and is literally the element "one" in `matrix I I α`. The element "one" is defined as the diagonal matrix induced by the vector of ones. The additional conditions for `matrix I I α` to have "one" are `[decidable_eq I] [has_zero α] [has_one α]`, where `[decidable_eq I]` means the equality relation over `I` is decidable. The conditions `[decidable_eq I] [has_zero α]` are needed by the function `diagonal` constructing diagonal matrices.

```
variables (I α : Type*) [fintype I]
variables [decidable_eq I] [has_zero α] [has_one α]
#check matrix.has_one
#check (1 : matrix I I α)
```

I implemented the matrices of all ones and "symmetric matrices" in file MATRIX_BASIC.LEAN:

```
/-- `matrix.all_one` is the matrix whose entries are all `1`s. -/
def all_one [has_one α]: matrix I J α := λ i, 1

/-- Proposition `matrix.is_sym`. `A.is_sym` means `Aᵀ = A`. -/
def is_sym (A : matrix I I α) : Prop := Aᵀ = A
```

and implemented relevant properties of symmetric matrices in file SYMMETRIC_MATRIX.LEAN, and also implemented "diagonal matrices" and relevant properties in file DIAGONAL_MATRIX.LEAN:

```
/-- `A.is_diagonal` means square matrix `A` is a dianogal matrix:
    `∀ i j, i ≠ j → A i j = 0`. -/
def is_diagonal [has_zero α] (A : matrix I I α) : Prop := ∀ i j, i ≠ j → A i j = 0
```

In the course of this project, I also implemented some other important concepts in linear algebra missing (or ever missing) in mathlib, which though are not widely used in the following chapters. The below lists several examples of them. Please see MATRIX_BASIC.LEAN for the complete code work for these concepts.

```
/-- `equiv.perm.to_matrix σ` is the permutation matrix given by
    a permutation `σ : equiv.perm I` on the index tpye `I`. -/
def equiv.perm.to_matrix [decidable_eq I] (α) [has_zero α] [has_one α] (σ : equiv.perm I) :
  matrix I I α
| i j := if σ i = j then 1 else 0

/-- The "trace" of a matrix.-/
def tr [add_comm_monoid α] (A : matrix I I α) : α := ∑ i : I, A i i

/-- The conjugate transpose of a matrix defined in term of `star`. -/
def conj_transpose [has_star α] (M : matrix I J α) : matrix J I α
| x y := star (M y x)
localized "postfix `ᴴ`:1500 := matrix.conj_transpose" in matrix

/-- Proposition `matrix.is_skewsym`. `A.is_skewsym` means `-Aᵀ = A` if `[has_neg α]`. -/
def is_skewsym [has_neg α] (A : matrix I I α) : Prop := -Aᵀ = A

/-- Proposition `matrix.is_Hermitian`. `A.is_Hermitian` means `Aᴴ = A` if `[has_star α]`. -/
def is_Hermitian [has_star α] (A : matrix I I α) : Prop := Aᴴ = A

/-- `matrix.similar_to` defines the proposition that matrix `A` is similar to `B`, denoted
    as `A ~ B`. -/
def similar_to (A B : matrix I I α) := ∃ (P : matrix I I α), is_unit P.det ∧ B = P⁻¹ * A * P
localized "notation `~`:50 := similar_to" in matrix

/-- Proposition `matrix.is_pos_def`. -/
def is_pos_def (M : matrix I I ℂ):= M.is_Hermitian ∧ ∀ v : I → ℂ, v ≠ 0 →
  0 < dot_product v (M.mul_vec v)
/-- Proposition `matrix.is_pos_semidef`. -/
def is_pos_semidef (M : matrix I I ℂ):= M.is_Hermitian∧ ∀ v : I → ℂ,
  0 ≤ dot_product v (M.mul_vec v)
/-- Proposition `matrix.is_neg_def`. -/
def is_neg_def (M : matrix I I ℂ):= M.is_Hermitian ∧ ∀ v : I → ℂ, v ≠ 0 →
  dot_product v (M.mul_vec v) < 0
/-- Proposition `matrix.is_neg_semidef`. -/
def is_neg_semidef (M : matrix I I ℂ):= M.is_Hermitian ∧ ∀ v : I → ℂ,
  dot_product v (M.mul_vec v) ≤ 0
```

3.2 Kronecker product

Definition 3.2.1. *If A is a matrix with (finite) row index set (index type in Lean) I and column index set J, and B is a matrix with row index set K and column index set L, then the* ***Kronecker product*** *$A \otimes B$ is the matrix with row index set $I \times K$ and column index set $J \times L$ and elements given by*

$$A \otimes B\ (i,\ k)\ (j,\ l) = (A\ i\ j) * (B\ k\ l),$$

for $(i,\ k) \in I \times K$ and $(j,\ l) \in J \times L$.

```
@[elab_as_eliminator]
def Kronecker [has_mul α] (A : matrix I J α) (B : matrix K L α) :
matrix (I × K) (J × L) α :=
λ ⟨i, k⟩ ⟨j, l⟩, (A i j) * (B k l)
```

```
/- an infix notation for the Kronecker product -/
localized "infix `⊗`:100 := matrix.Kronecker" in matrix
```

The implementation of the Kronecker product uses the identifier “Kronecker” for short. The tag `@[elab_as_eliminator]` guides Lean’s elaborator about how to elaborate this function.

We next implement some basic properties of the Kronecker product, such as the distributivity, associativity, and the mixed-product properties. Here, we list some of them (with implemented proofs omitted). Please see MAIN1.LEAN for the complete implementation.

```
lemma Kronecker_apply [has_mul α]
(A : matrix I J α) (B : matrix K L α) (a : I × K) (b : J × L) :
(A ⊗ B) a b = (A a.1 b.1) * (B a.2 b.2)

/- distributivity over addition -/
section distrib
variables [distrib α] -- variables are restricted to this section
variables (A : matrix I J α) (B : matrix K L α) (B' : matrix K L α)
lemma K_add :A ⊗ (B + B') = A ⊗ B + A ⊗ B'
lemma add_K :(B + B') ⊗ A = B ⊗ A + B' ⊗ A
end distrib

/- distributivity over substraction -/
section distrib_sub
variables [ring α]
variables (A : matrix I J α) (B : matrix K L α) (B' : matrix K L α)
lemma K_sub :A ⊗ (B - B') = A ⊗ B - A ⊗ B'
lemma sub_K :(B - B') ⊗ A = B ⊗ A - B' ⊗ A
end distrib_sub

/-- associativity -/
lemma K_assoc
[semigroup α] (A : matrix I J α) (B : matrix K L α) (C : matrix M N α) :
A ⊗ B ⊗ C = A ⊗ (B ⊗ C)

section zero
variables [mul_zero_class α] (A : matrix I J α)
@[simp] lemma K_zero : A ⊗ (0 : matrix K L α) = 0
@[simp] lemma zero_K : (0 : matrix K L α) ⊗ A = 0
end zero

/-- `1 ⊗ 1 = 1`.
    The Kronecker product of two identity matrices is an identity matrix. -/
@[simp] lemma one_K_one
[mul_zero_one_class α] [decidable_eq I] [decidable_eq J] :
(1 :matrix I I α) ⊗ (1 :matrix J J α) = 1

section neg
variables [ring α]
variables (A : matrix I J α) (B : matrix K L α)
@[simp] lemma neg_K: (-A) ⊗ B = - A ⊗ B
@[simp] lemma K_neg: A ⊗ (-B) = - A ⊗ B
end neg
```

```
/- scalar multiplication -/
section scalar
@[simp] lemma smul_K
[has_mul α] [has_scalar R α] [is_scalar_tower R α α]
(k : R) (A : matrix I J α) (B : matrix K L α) :
(k • A) ⊗ B = k • A ⊗ B
@[simp] lemma K_smul
[has_mul α] [has_scalar R α] [smul_comm_class R α α]
(k : R) (A : matrix I J α) (B : matrix K L α) :
A ⊗ (k • B) = k • A ⊗ B
end scalar

lemma transpose_K
[has_mul α] (A : matrix I J α) (B : matrix K L α):
(A ⊗ B)ᵀ = Aᵀ ⊗ Bᵀ

lemma conj_transpose_K
[comm_monoid α] [star_monoid α] (M₁ : matrix I J α) (M₂ : matrix K L α) :
(M₁ ⊗ M₂)ᴴ = M₁ᴴ ⊗ M₂ᴴ

/- Kronecker product mixes matrix multiplication -/
section Kronecker_mul
variables [comm_ring α]
variables
(A : matrix I J α) (C : matrix J K α)
(B : matrix L M α) (D : matrix M N α)
lemma K_mul: (A ⊗ B) ⬝ (C ⊗ D) = (A ⬝ C) ⊗ (B ⬝ D)
end Kronecker_mul

/- Kronecker product mixes Hadamard product -/
section Kronecker_Hadamard
variables [comm_semigroup α]
(A : matrix I J α) (C : matrix I J α)
(B : matrix K L α) (D : matrix K L α)
lemma Kronecker_Hadamard : (A ⊗ B) ⊙ (C ⊗ D) = (A ⊙ C) ⊗ (B ⊙ D)
end Kronecker_Hadamard

section inverse
variables [decidable_eq I] [decidable_eq J] [comm_ring α]
variables (A : matrix I I α) (B : matrix J J α) (C : matrix I I α)
lemma K_inverse [invertible A] [invertible B] :(A ⊗ B)⁻¹ = A⁻¹ ⊗ B⁻¹
end inverse

/-- `A ⊗ B` is symmetric if `A` and `B` are symmetric. -/
lemma is_sym_K_of [has_mul α] {A : matrix I I α} {B : matrix J J α}
(ha : A.is_sym) (hb : B.is_sym) : (A ⊗ B).is_sym
```

The names and statements of such lemmas are usually self-explaining. For example, `one_K_one` proves that the Kronecker product of two identity matrices is also an identity matrix. The "K" in the identifier is an abbreviation for "Kronecker product". For another example, `is_sym_K_of` proves that if `(ha : A.is_sym) (hb : B.is_sym)` (i.e. `A`, `B` are symmetric), then `(A ⊗ B).is_sym`. The "of" in the identifier signals that the conclusion follows from certain hypotheses.

The difficulty of implementing these basic properties usually does not come from the poofs themselves. The difficulty is to figure out the minimum assumptions required for a property to hold to make the implemented version as general as possible. For example, for `1 ⊗ 1 = 1`, where entries are in `α`, to hold, the minimum conditions are `[mul_zero_one_class α] [decidable_eq I] [decidable_eq J]`, where `mul_zero_one_class` is the typeclass for non-associative monoids with zero elements. For another example, for matrices with entries in `α`, the minimum condition for the distributivity of the Kronecker product over addition is `[distrib α]`, where `distrib` is a typeclass stating that the multiplication is left and right distributive over addition. While the minimum condition for the distributivity over subtraction is `[ring α]` (i.e. `α` is a ring) based on the current infrastructure of mathlib. The minimum assumptions required for implementing a certain property are influenced or determined by both the mathematics behind and the current infrastructure of mathlib (how the ingredients used to formalise this properties are currently encoded in mathlib).

Note that we proved two mixed-product properties: one is the mixed-product of the Kronecker product and the matrix multiplication; one is the mixed-product of the Kronecker product and the Hadamard product. The Hadamard product is not implemented anywhere in mathlib. We will discusses its implementation briefly in the next section.

One primary mission for writing mathlib is to build proper and enough application programming interfaces (APIs) for every enough trivial definition. Such work not only allows other end users to conveniently use and invoke built definitions and lemmas, but also helps us to prove more involved lemma/theorems and makes the poofs of "big" theorems neat and straightforward. An API can just be a basic maths property such as `transpose_K`, which proves `(A ⊗ B)ᵀ = Aᵀ ⊗ Bᵀ`. An API can also be a lemma which is mathematically trivial such as `Kronecker_apply`, which proves `(A ⊗ B) a b = (A a.1 b.1) * (B a.2 b.2)`; but with this API built, rewrite tactics, such as `rw`, are then able to invoke it to rewrite or simplify terms for us in future proofs. Another such example is

```
lemma is_sym.eq {A : matrix I I α} (h : A.is_sym) : Aᵀ = A := h
```

in SYMMETRIC_MATRIX.LEAN, which enables a rewrite tactic to rewrite `Aᵀ` to `A` if `A` is symmetric. The lemmas/APIs labelled by `simp` will also enhance the the Lean simplifier.

We end this section by a final remark. Did you notice that we implemented the associativity that `A ⊗ B ⊗ C = A ⊗ (B ⊗ C)`? Apologises! I lied to you. The formulation is a lie as Lean does not allow to compare two expressions of different types by the equality relation. Lean will reports an error if the two sides of "=" do not have the same type. Indeed, `A ⊗ B ⊗ C` has type `matrix ((I × K) × M) ((J × L) × N) α`, and `A ⊗ (B ⊗ C)` has type `matrix (I × (K × M)) (J × (L × N)) α`, and the two types are not definitionally the same! In face of this problem, you have two options: (1) give up formalising the associativity, and complain Lean is dumb; (2) try harder to implement it somehow just like me! We note that there is a natural equivalence between `(I × K) × M` and `I × (K × M)`. The truth behind the given formulation is actually `A ⊗ B ⊗ C = ↑(A ⊗ (B ⊗ C))`, where the arrow coerces `A ⊗ (B ⊗ C)` into the type `matrix ((I × K) × M) ((J × L) × N) α`. In this case, Lean is satisfied with the formulation `A ⊗ B ⊗ C = A ⊗ (B ⊗ C)` rather than shouting at me, because I have already secretly built all the required coercions in a dark corner and hinted Lean to elaborate this equation properly by tagging `@[elab_as_eliminator]` to the definition of the Kronecker product, such that Leans knows to insert the arrow itself when parsing through.

3.3 Interlude: Hadamard product

Definition 3.3.1. *For two matrices A and B of the same dimension $m \times n$, the **Hadamard product** (also known as the element-wise product, entrywise product, or Schur product) $A \odot B$ is a matrix of the same dimension as the operands, with elements given by $A \odot B\ i\ j = (A\ i\ j) * (B\ i\ j)$.*

```
def Hadamard [has_mul α] (A : matrix I J α) (B : matrix I J α) :
matrix I J α :=
λ i j, (A i j) * (B i j)
localized "infix `⊙`:100 := matrix.Hadamard" in matrix -- declares the notation
```

We implement some properties of Hadamard product:

```
variables (A : matrix I J α) (B : matrix I J α) (C : matrix I J α)

/- commutativity -/
lemma Had_comm [comm_semigroup α] : A ⊙ B = B ⊙ A

/- associativity -/
lemma Had_assoc [semigroup α] : A ⊙ B ⊙ C = A ⊙ (B ⊙ C)

/- distributivity -/
section distrib
variables [distrib α]
lemma Had_add : A ⊙ (B + C) = A ⊙ B + A ⊙ C
lemma add_Had : (B + C) ⊙ A = B ⊙ A + C ⊙ A
end distrib

/- scalar multiplication -/
section scalar
@[simp] lemma smul_Had
[has_mul α] [has_scalar R α] [is_scalar_tower R α α] (k : R) :
(k • A) ⊙ B = k • A ⊙ B
@[simp] lemma Had_smul
[has_mul α] [has_scalar R α] [smul_comm_class R α α] (k : R):
A ⊙ (k • B) = k • A ⊙ B
end scalar

section zero
variables [mul_zero_class α]
@[simp] lemma Had_zero : A ⊙ (0 : matrix I J α) = 0
@[simp] lemma zero_Had : (0 : matrix I J α) ⊙ A = 0
end zero

section trace
variables [comm_semiring α] [decidable_eq I] [decidable_eq J]

/--`vᵀ (M₁ ⊙ M₂) w = tr ((diagonal v)ᵀ · M₁ · (diagonal w) · M₂ᵀ)` -/
lemma tr_identity (v : I → α) (w : J → α) (M₁ : matrix I J α) (M₂ : matrix I J α):
dot_product (vec_mul  v  (M₁ ⊙ M₂)) w =
tr ((diagonal v)ᵀ · M₁ · (diagonal w) · M₂ᵀ)

/-- `∑ (i : I) (j : J), (M₁ ⊙ M₂) i j = tr (M₁ · M₂ᵀ)` -/
lemma sum_Had_eq_tr_mul (M₁ : matrix I J α) (M₂ : matrix I J α) :
∑ (i : I) (j : J), (M₁ ⊙ M₂) i j = tr (M₁ · M₂ᵀ)
```

```
/-- `v^H (M₁ ⊙ M₂) w = tr ((diagonal v)^H · M₁ · (diagonal w) · M₂ᵀ)` over `ℂ` -/
lemma tr_identity_over_ℂ
(v : I → ℂ) (w : J → ℂ) (M₁ : matrix I J ℂ) (M₂ : matrix I J ℂ):
dot_product (vec_mul (star v)  (M₁ ⊙ M₂)) w =
tr ((diagonal v)^H · M₁ · (diagonal w) · M₂ᵀ)
end trace
```

4 QUADRATIC RESIDUES AND QUADRATIC CHARACTERS

Paley's constructions of Hadamard matrices are based on finite fields and use the notions of quadratic residues and the quadratic character. As a preparation, this chapter discusses the implementation of quadratic residues, the quadratic character, and some other relevant results about finite fields.

- F denotes a finite field, and p denotes the characteristic of F, and q denotes the cardinality of F. F may also be denoted as $GF(q)$. F^* denotes the unit group of F.
- The implementation in this chapter does not follow any particular materials. The definition of the quadratic character is excerpted from [23].
- Please see file FINITE_FIELD.LEAN for the complete code work of this chapter's topic.

4.1 Finite fields

This section introduces how basic features of fields and finite fields are implemented in mathlib and supplies results that are not in mathlib but required in the following sections.

The notion "field" is implemented as a type class in mathlib, while the details (provided below) of how this type class is implemented doesn't matter very much, as what an end user will use are the built APIs of this definition.

```
/-- A `field` is a `comm_ring` with multiplicative inverses for nonzero elements -/
@[protect_proj, ancestor comm_ring div_inv_monoid nontrivial]
class field (K : Type u) extends comm_ring K, div_inv_monoid K, nontrivial K :=
(mul_inv_cancel : ∀ {a : K}, a ≠ 0 → a * a⁻¹ = 1)
(inv_zero : (0 : K)⁻¹ = 0)
```

Mathematically, a finite field (also called a Galois field) is a field that has a finite number of elements. The assumption that some F is a finite field splits into the conditions `{F : Type*} [field F] [fintype F]` in Lean. A natural number p is the characteristic (char) of the finite field F is implemented as `{p : ℕ} [char_p F p]`, where `char_p` is a type class encoded as

```
class char_p (R : Type u) [add_monoid R] [has_one R] (p : ℕ) : Prop :=
(cast_eq_zero_iff [] : ∀ x:ℕ, (x:R) = 0 ↔ p ∣ x)
```

Thus, `[char_p F p]` implies `∀ x : ℕ, (x : F) = 0 ↔ p ∣ x)`.

In mathlib, `fintype.card`, which is in turn defined on `finset.card`, is the function that outputs the cardinality of a finite type.

For an element x in a (multiplicative) monoid (in particular, x can be in a field), the (multiplicative) order of x, denoted as $o(x)$ is the smallest $n \geqslant 1$ such that $x^n = 1$ if such n exists; and is 0 if such n does not exist, in which case we call x is of infinite order. In mathlib, `order_of x` is the function that outputs the order of `x`, and is implemented as:

```
noncomputable def order_of {G : Type u} [monoid G] (x : G) : ℕ :=
minimal_period ((*) x) 1
```

We fix F to be a finite field with char p and cardinality q, and set the Lean environment as follows from now on in this chapter.

```
namespace finite_field
variables {F : Type*} [field F] [fintype F] {p : ℕ} [char_p F p]
local notation `q` := fintype.card F -- declares `q` as a notation
```

The unit group F^* of F is the group of units in F. As F is a field, it contains all the elements except 0. F^* in Lean is expressed as `units F`, where `units` is implemented as:

```
/-- Units of a monoid. An element of a `monoid` is a unit if it has a two-sided
inverse. This version bundles the inverse element so that it can be computed. For a predicate
```

```
see `is_unit`. -/
structure units (α : Type u) [monoid α] :=
(val : α)
(inv : α)
(val_inv : val * inv = 1)
(inv_val : inv * val = 1)
```

Thus, an element in `units F` is, by definition, not an element in `F`, but whose fist field is. Mathlib has a coercion that transfers an element in `units F` to the corresponding element in `F`, and conversely, I built `to_unit`, which converts a non-zero element in `F` into `units F`.

```
/-- For non-zero `a : F`, `to_unit` converts `a` to an instance of `units F`. -/
@[simp] def to_unit {a : F} (h : a ≠ 0): units F :=
by refine {val := a, inv := a⁻¹, ..}; simp* at *
```

The `refine` tactic is able to split the goal of constructing an expression of some structure into subgoals of constructing each field of this structure. `finite_field.card_units` in mathlib is the proof that `fintype.card (units F) = fintype.card F - 1`.

We contribute several new lemmas about finite fields that we will use.

Lemma 4.1.1. *If $p \neq 2$ and $x \in F$, $o(x) = 2 \Leftrightarrow x = -1$*

```
lemma order_of_eq_two_iff (hp: p ≠ 2) (x : F) : order_of x = 2 ↔ x = -1

/-- The "units" version of `order_of_eq_two_iff`. -/
lemma order_of_eq_two_iff' (hp: p ≠ 2) (x : units F) : order_of x = 2 ↔ x = -1
```

Lemma 4.1.2. *If $p \neq 2$, -1 is a square in $F \Leftrightarrow q \equiv 1 \pmod 4$*

Proof. Suppose $p \neq 2$. As $|F^*| = q - 1$, it suffices to show -1 is a square $\Leftrightarrow 4 \mid |F^*|$.

- $\Rightarrow$ Suppose $-1 = a^2$. Then $a^4 = 1$, and thus $o(a) \mid 4$, and thus $o(a) \in \{1, 2, 4\}$. As $a \neq 1$, we have $o(a) \neq 1$. As $a \neq -1$ we have $o(a) \neq 2$ by Lemma 4.1.1. Thus, $o(a) = 4$. Since $a \in F^*$, $4 = o(a) \mid |F^*|$.
- $\Leftarrow$ Suppose $|F^*| = 4k$ for some $k \in \mathbb{N}$. As F^* is a cyclic group, there exists generator some $g \in F^*$ such that $F^* = \langle g \rangle$. As $g^{|F^*|} = g^{4k} = (g^{2k})^2 = 1$, $g^{2k} \in \{1, -1\}$. Suppose to the contrary $g^{2k} = 1$ so that $o(g) \leq 2k$. As g is the generator of F^*, $o(g) = |F^*| = 4k$. As $|F^*| \neq 0$, $k > 0$. Then $4k = o(g) \leq 2k$ is a contradiction. Thus, $g^{2k} = -1$ and $-1 = (g^k)^2$.

In the implemented proof,

- `order_of_dvd_of_pow_eq_one` proves that, for any $a \in F^*$, if $a^n = 1$, then $o(a) \mid n$,
- `order_of_dvd_card_univ` proves that $o(a) \mid |F^*|$,
- `is_cyclic.exists_generator` proves that there is a generator g generating the cyclic group F^*,
- `pow_card_eq_one` proves that, for any $a \in F^*$, $a^{|F^*|} = 1$,
- `order_of_eq_card_of_forall_mem_gpowers` proves if g generates a finite group G, then $o(g) = |G|$.

`neg_one_eq_sq_iff'` views -1 as an element in `units F`, and `neg_one_eq_sq_iff` encapsulates it.

```
/-- `-1` is a square in `units F` iff the cardinal `q ≡ 1 [MOD 4]`. -/
theorem neg_one_eq_sq_iff' (hp: p ≠ 2) :
(∃ a : units F, -1 = a^2) ↔ q ≡ 1 [MOD 4] :=
begin
  -- rewrites the RHS to `4 | fintype.card (units F)`
  rw card_eq_one_mod_n_iff_n_divide_card_units,
  split, -- splits the goal into two directions
  -- the `→` direction
```

```
  { rintros ⟨a, h'⟩,
    -- h: a ^ 4 = 1
    have h := calc a^4 = a^2 * a^2 : by group
                ... = 1: by simp [← h'],
    -- h₀: order_of a | 4
    have h₀ := order_of_dvd_of_pow_eq_one h,
    -- au: order_of a ≤ 4
    have au := nat.le_of_dvd dec_trivial h₀,
    -- g₁ : a ≠ 1
    have g₁ : a ≠ 1,
    { rintro rfl, simp at h',
      exact absurd h' (neg_one_ne_one_of' hp) },
    -- h₁ : order_of a ≠ 1
    have h₁ := mt order_of_eq_one_iff.1 g₁,
    -- g₂ : a ≠ -1
    have g₂ : a ≠ -1,
    { rintro rfl, simp [pow_two] at h',
      exact absurd h' (neg_one_ne_one_of' hp) },
    -- h₂ : order_of a ≠ 2
    have h₂ := mt (order_of_eq_two_iff' hp a).1 g₂,
    -- ha : order_of a = 4
    have ha : order_of a = 4,
    { revert h₀ h₁ h₂,
      interval_cases (order_of a),
      any_goals {rw h_1, norm_num} },
    simp [← ha, order_of_dvd_card_univ] },
  -- the `←` direction
  { rintro ⟨k, hF⟩, -- hF : |units F| = 4 * k
    -- `hg'` is a proof that `g` generates `units F`
    obtain ⟨g, hg'⟩ := is_cyclic.exists_generator (units F),
    -- hg : g ^ |units F| = 1
    have hg := @pow_card_eq_one (units F) g _ _,
    have eq : 4 * k = k * 2 * 2, {ring},
    -- rewrite `hg` to `hg : g ^ (k * 2) = 1 ∨ g ^ (k * 2) = -1`
    rw [hF, eq, pow_mul, sq_eq_one_iff_eq_one_or_eq_neg_one'] at hg,
    rcases hg with (hg | hg), -- splits into two cases
    -- case `g ^ (k * 2) = 1`
    { have hk₁ := card_pos_iff.mpr ⟨(1 : units F)⟩,
      rw hF at hk₁,
      have o₁ := order_of_eq_card_of_forall_mem_gpowers hg',
      have o₂ := order_of_dvd_of_pow_eq_one hg,
      have le := nat.le_of_dvd (by linarith) o₂,
      rw [o₁, hF] at le,
      exfalso, linarith },
    -- case `g ^ (k * 2) = -1`
    { use g ^ k, simp [← hg, pow_mul] } },
end

/-- `-1` is a square in `F` iff the cardinal `q ≡ 1 [MOD 4]`. -/
lemma neg_one_eq_sq_iff (hp: p ≠ 2) :
(∃ a : F, -1 = a^2) ↔ fintype.card F ≡ 1 [MOD 4] :=
begin
```

```
  rw [←neg_one_eq_sq_iff' hp],
  -- the current goal is
  -- `(∃ (a : F), -1 = a ^ 2) ↔ ∃ (a : units F), -1 = a ^ 2`
  split, -- splits into two directions
  any_goals {rintros ⟨a, ha⟩},
  -- the `→` direction
  { have ha' : a ≠ 0,
    {rintros rfl, simp* at *},
    use (to_unit ha'),
    simpa [units.ext_iff] },
  -- the `←` direction
  { use a, simp [units.ext_iff] at ha, assumption },
  assumption
end
```

■

4.2 Quadratic residues

Definition 4.2.1. *For a non-zero element a in F, we say a is a* ***quadratic residue*** *(abbr. residue) if $\exists\, b,\ a = b^2$, and say a is a* ***quadratic non-residue*** *(abbr. non-residue) otherwise.*
The definition is implemented as two propositions:

```
/-- A proposition predicating whether a given `a : F` is a quadratic residue in `F`.
    `finite_field.is_quad_residue a` is defined to be `a ≠ 0 ∧ ∃ b, a = b^2`. -/
def is_quad_residue (a : F) : Prop := a ≠ 0 ∧ ∃ b, a = b^2

/-- A proposition predicating whether a given `a : F` is a quadratic non-residue in `F`.
    `finite_field.is_non_residue a` is defined to be `a ≠ 0 ∧ ¬ ∃ b, a = b^2`. -/
def is_non_residue (a : F) : Prop := a ≠ 0 ∧ ¬ ∃ b, a = b^2
```

When equality over `F` is decidable, the two propositions are decidable. We implement this fact as two instances:

```
instance [decidable_eq F] (a : F) : decidable (is_quad_residue a) :=
by {unfold is_quad_residue, apply_instance}

instance [decidable_eq F] (a : F) : decidable (is_non_residue a) :=
by {unfold is_non_residue, apply_instance}
```

We will use subtypes `{a : F // is_quad_residue a}` and `{a : F // is_non_residue a}` of `F` to denote the "sets" of residues and non-residues in F, where "subtype" is implemented a structure family in mathlib:

```
structure subtype {α : Sort u} (p : α → Prop) :=
(val : α) (property : p val)
```

That is, an element `a` in the subtype of `α` induced by a predicate `p` (the subtype is denoted as `subtype p` or `{a : α // p a}` in mathlib) consists of two fields: `a.val` and `a.property`, and `a.property` proves that `p a.val` is true. Mathlib implemented a coercion from `a` to `a.val`, and an instance of `[fintype subtype p]` if `p` is a decidable predicate.

In the proofs in words, we use R and N to denote `{a : F // is_quad_residue a}` and `{a : F // is_non_residue a}` respectively, and may call R and N "sets". It is trivial to see $q = 1+|R|+|S|$. We will also see, if $p \neq 2$, then $|R| = |N|$ (when $p = 2$, one can show $|R| = q - 1$ and $|N| = 0$).

To establish $p \neq 2$ implies $|R| = |N|$, we first define the following auxiliary map:

Definition 4.2.2. *Define* $sq : F^* \to F^*$ *by* $sq(a) = a * a$.
sq is implemented as a bundled group homomorphism `sq`, which takes `F` as an explicit parameter:

```
variables (F) -- re-declares `F` as an explicit variable
/-- `sq` is the square function from `units F` to `units F`,
    defined as a group homomorphism. -/
def sq : (units F) →* (units F) :=
⟨λ a, a * a, by simp, (λ x y, by simp [units.ext_iff]; ring)⟩
```

Here, "bundled" means `→*` is implemented in mathlib as a structure, whose first field is the map itself and other fields are the properties the map satisfies. The below shows the exact implementation of `→*`:

```
/-- Bundled monoid homomorphisms; use this for bundled group homomorphisms too. -/
structure monoid_hom (M : Type*) (N : Type*) [mul_one_class M] [mul_one_class N]
  extends one_hom M N, mul_hom M N
```

The first group isomorphism theorem (implemented in mathlib as `quotient_ker_equiv_range`) implies that

Lemma 4.2.3. $|F^*| = |range\ sq| * |ker\ sq|$

```
/-- `|units F| = |(sq F).range| * |(sq F).ker|` -/
theorem sq.iso [decidable_eq F] :
fintype.card (units F) = fintype.card (sq F).range * fintype.card (sq F).ker
```

We note that the range of *sq* is exactly the residues R of F, and when $p \neq 2$, the kernel of *sq* is exactly $\{1, -1\}$, where $1 \neq -1$. Therefore, we have:

Lemma 4.2.4. *If* $p \neq 2$, *then* $|F^*| = |R| * 2$.

```
/-- `|units F| = |{a : F // is_quad_residue a}| * 2` -/
theorem card_units_eq_card_residues_mul_two [decidable_eq F] (hp: p ≠ 2) :
fintype.card (units F) = fintype.card {a : F // is_quad_residue a} * 2
```

Combine Lemma 4.2.4, $|F^*| = q - 1$, and $q = 1 + |R| + |S|$, we can finally establish that

Lemma 4.2.5. *If* $p \neq 2$, F *has an equal number of residues and non-residues, i.e.* $|R| = |N|$.

```
/-- `|{a : F // is_quad_residue a}| = |{a : F // is_non_residue a}|`-/
theorem card_residues_eq_card_non_residues
[decidable_eq F] (hp : p ≠ 2):
fintype.card {a : F // is_quad_residue a} =
fintype.card {a : F // is_non_residue a} :=
begin
  have eq := eq_one_add_card_residues_add_card_non_residues F,
  simp [card_units', card_units_eq_card_residues_mul_two F hp] at eq,
  linarith
end
/- `card_units_eq_card_residues_mul_two F hp` is a built API that proves
   `|F| = 1 + |{a : F // is_quad_residue a}| + |{a : F // is_non_residue a}|` -/

/-- unfolded version of `card_residues_eq_card_non_residues` -/
theorem card_residues_eq_card_non_residues'
[decidable_eq F] (hp : p ≠ 2):
(@univ {a : F // is_quad_residue a} _).card =
(@univ {a : F // is_non_residue a} _).card :=
by convert card_residues_eq_card_non_residues F hp
```

In the implementation, `card_residues_eq_card_non_residues'` is the version that unfolds the implemented definition of `fintype.card`.

As when $p \neq 2$, -1 is a square in F if and only if $q \equiv 1 \pmod 4$, proved by Lemma 4.1.2, we have the following results:

Lemma 4.2.6.
(1) If $q \equiv 1 \pmod 4$, -1 is a residue in F.
(2) If $q \equiv 3 \pmod 4$, -1 is a non-residue in F.

```
variable {F} -- re-declares `F` back to an implicit variable
/-- `-1` is a residue if `q ≡ 1 [MOD 4]`. -/
lemma neg_one_is_residue_of (hF : q ≡ 1 [MOD 4]) :
is_quad_residue (-1 : F)

/-- `-1` is a non-residue if `q ≡ 3 [MOD 4]`. -/
lemma neg_one_is_non_residue_of (hF : q ≡ 3 [MOD 4]) :
is_non_residue (-1 : F)
```

We end this section by two more lemmas:

Lemma 4.2.7. *Suppose a is in F (a^{-1} is defined to be 0 in mathlib for $a = 0$).*
(1) a^{-1} is a residue $\Leftrightarrow$ a is a residue.
(2) a^{-1} is a non-residue $\Leftrightarrow$ a is a non-residue.

Proof. We will prove (1), and (2) follows from (1).

- If $a^{-1} = b^2$ for some $b \neq 0$, then $a = (b^{-1})^2$ and $a \neq 0$.
- If $a = b^2$ for some $b \neq 0$, then $a^{-1} = (b^{-1})^2$ and $a \neq 0$.

```
/-- `a⁻¹` is a residue if and only if `a` is. -/
theorem inv_is_residue_iff {a : F} :
is_quad_residue a⁻¹ ↔ is_quad_residue a :=
begin
  split, -- splits into two directions
  any_goals {rintro ⟨h, b, g⟩, refine ⟨by tidy, b⁻¹, by simp [←g]⟩},
end

/-- `a⁻¹` is a non residue if and only if `a` is. -/
theorem inv_is_non_residue_iff {a : F} :
is_non_residue a⁻¹ ↔ is_non_residue a :=
begin
  by_cases h : a = 0,
  {simp* at *}, -- when `a = 0`
  have h' : a⁻¹ ≠ 0 := by simp [h],
  simp [←not_residue_iff_is_non_residue, *, inv_is_residue_iff]
end
```

■

Lemma 4.2.8. *Suppose a, b are in F.*
(1) If a, b are residues, then $a * b$ is a residue.
(2) If a is a non-residue and b is a residues, then $a * b$ is a non-residue.
(3) If a is a residue and b is a non-residues, then $a * b$ is a non-residue.
(4) If a, b are non-residues and $p \neq 2$, then $a * b$ is a residue.

Proof. Suppose a, b are in F.
(1) If $a = c^2$ and $c \neq 0$, and $b = d^2$ and $d \neq 0$, then $ab = (cd)^2$ and $cd \neq 0$.

(2) If a is a non-residue, and $b = c^2$ and $c \neq 0$, suppose, for a contradiction, $ab = d^2$ and $d \neq 0$. Then $a = (dc^{-1})^2$ is a residue.
(3) If a is a residue and b is a non-residues, $ab = ba$ is a non-residue by (2).
(4) If a, b are non-residues and $p \neq 2$, we define $f : R \to N$ to be the function $a * \_$.
As $ac = ad$ implies $c = d$, f is injective. Because f is injective and $|R| = |N|$ by Lemma 4.2.5, f is surjective.
Suppose, for a contradiction, $a * b$ is a non-residue. As f is surjective, there is some residue b' such that $a * b' = a * b$. As $a \neq 0$, $b' = b$. b is a non-residue and b' is a residue yields a contradiction.

```
theorem residue_mul_residue_is_residue
{a b : F} (ha : is_quad_residue a) (hb : is_quad_residue b) :
is_quad_residue (a * b) :=
begin
  obtain ⟨ha, c, rfl⟩ := ha,
  obtain ⟨hb, d, rfl⟩ := hb,
  refine ⟨mul_ne_zero ha hb, c*d, _⟩,
  ring
end

theorem non_residue_mul_residue_is_non_residue
{a b : F} (ha : is_non_residue a) (hb : is_quad_residue b) :
is_non_residue (a * b) :=
begin
  obtain ⟨hb, c, rfl⟩ := hb,
  refine ⟨mul_ne_zero ha.1 hb, _⟩,
  rintro ⟨d, h⟩,
  convert ha.2 ⟨(d * c⁻¹), _⟩,
  field_simp [← h], -- `field_simp` is the specialised version of `simp` for `field`
end

theorem residue_mul_non_residue_is_non_residue
{a b : F} (ha : is_quad_residue a) (hb : is_non_residue b):
is_non_residue (a * b) :=
by simp [mul_comm a, non_residue_mul_residue_is_non_residue hb ha]

/-- `finite_filed.non_residue_mul` is the map `a * _` given a non-residue `a`
    defined on `{b : F // is_quad_residue b}`. -/
def non_residue_mul {a: F} (ha : is_non_residue a) :
{b : F // is_quad_residue b} → {b : F // is_non_residue b} :=
λ b, ⟨a * b, non_residue_mul_residue_is_non_residue ha b.2⟩

open function

/-- proves that `a * _` is injective from residues for a non-residue `a` -/
lemma is_non_residue.mul_is_injective
{a: F} (ha : is_non_residue a) :
injective (non_residue_mul ha):=
begin
  intros b₁ b₂ h,
  simp [non_residue_mul] at h,
  ext,
  convert or.resolve_right h ha.1,
```

```
end

/-- proves that `a * _` is surjective onto non-residues for a non-residue `a` -/
lemma is_non_residue.mul_is_surjective
[decidable_eq F] (hp : p ≠ 2) {a: F} (ha : is_non_residue a) :
surjective (non_residue_mul ha):=
begin
  by_contra, -- prove by contradtiction
  have lt := card_lt_of_injective_not_surjective
    (non_residue_mul ha) (ha.mul_is_injective) h,
  have eq := card_residues_eq_card_non_residues F hp,
  linarith
end

theorem non_residue_mul_non_residue_is_residue
[decidable_eq F] (hp : p ≠ 2)
{a b : F} (ha : is_non_residue a) (hb : is_non_residue b):
is_quad_residue (a * b) :=
begin
  by_contra h, -- prove by contradtiction
  -- rw `h` to `is_non_residue (a * b)`
  rw [not_residue_iff_is_non_residue (mul_ne_zero ha.1 hb.1)] at h,
  -- surj : `a * _` is surjective onto non-residues from residues
  have surj := ha.mul_is_surjective hp,
  simp [function.surjective] at surj,
  -- in particular, non-residue `a * b` is in the range of `a * _`
  specialize surj (a * b) h,
  -- say `a * b' = a * b` and `hb' : is_quad_residue b'`
  rcases surj with ⟨b', hb', eq⟩,
  simp [non_residue_mul, ha.1] at eq, -- `eq: b' = b`
  rw [eq] at hb',
  exact absurd hb'.2 hb.2,
end
```

■

Remark. The condition $p \neq 2$ in case (4) is redundant. We can git rid of it by giving a further discussion for when $p = 2$. As in Paley's constructions, $p \neq 2$ is a necessary assumption, this theorem here is general enough for our purpose.

4.3 Quadratic character

In the section, we assume the decidability of the equality relation over F, which is a necessary condition for implementing the definition of the quadratic character.

Definition 4.3.1. *For $a \in F$, the **quadratic character** $\chi(a)$ of a is 0 if $a = 0$, and is 1 if a is a residue, and is -1 if a is a non-residue.*

```
variables {F} [decidable_eq F]
def quad_char (a : F) : ℚ :=
if a = 0            then 0 else
if ∃ b : F, a = b^2 then 1 else
                         -1.
notation `χ` := quad_char  -- declare the notation for `quad_char`
```

We implement `χ a` to have type `ℚ`, as we will implement entries of a Hadamard matrix to have type `ℚ`, the reason for which will be explained when they are implemented.

Many results about quadratic characters follows from what we have established about quadratic residues immediately: for examples, Lemma 4.2.7 implies

LEMMA 4.3.2. *For a in F, $\chi(a^{-1}) = \chi(a)$*

```
theorem quad_char_inv (a : F) : χ a⁻¹ = χ a :=
begin
  by_cases ha: a = 0,
  {simp [ha]}, -- case `a=0`
  -- splits into cases if `a` is a residue
  obtain (ga | ga) := residue_or_non_residue ha,
  -- case 1: `a` is a residue
  {have g':= inv_is_residue_iff.2 ga, simp*},
  -- case 2: `a` is a non-residue
  {have g':= inv_is_non_residue_iff.2 ga, simp*},
end
```

and Lemma 4.2.8 implies

LEMMA 4.3.3. *If $p \neq 2$, for a, b in F, $\chi(ab) = \chi(a)\chi(b)$*

```
theorem quad_char_mul (hp : p ≠ 2) (a b : F) : χ (a * b) = χ a * χ b :=
begin
  by_cases ha: a = 0,
  any_goals {by_cases hb : b = 0},
  any_goals {simp*}, -- closes cases when `a = 0` or `b =0`
  -- splits into cases if `a` is a residue
  obtain (ga | ga) := residue_or_non_residue ha,
  -- splits into cases if `b` is a residue
  any_goals {obtain (gb | gb) := residue_or_non_residue hb},
  -- case 1 : `a` is, `b` is
  { have g:= residue_mul_residue_is_residue ga gb, simp* },
  -- case 2 : `a` is, `b` is not
  { have g:= residue_mul_non_residue_is_non_residue ga gb, simp* },
  -- case 1 : `a` is not, `b` is
  { have g:= non_residue_mul_residue_is_non_residue ga gb, simp* },
  -- case 4 : `a` is not, `b` is not
  { have g:= non_residue_mul_non_residue_is_residue hp ga gb, simp* },
end
```

and Lemma 4.2.6 implies

LEMMA 4.3.4.

(1) If $q \equiv 1 \pmod 4$, $\chi(-1) = 1$.

(2) If $q \equiv 3 \pmod 4$, $\chi(-1) = -1$.

```
/-- `χ (-1) = 1` if `q ≡ 1 [MOD 4]`. -/
@[simp] theorem char_neg_one_eq_one_of (hF : q ≡ 1 [MOD 4]) :
χ (-1 : F) = 1 :=
by simp [neg_one_is_residue_of hF]

/-- `χ (-1) = -1` if `q ≡ 3 [MOD 4]`. -/
@[simp] theorem char_neg_one_eq_neg_one_of (hF : q ≡ 3 [MOD 4]) :
χ (-1 : F) = -1 :=
by simp [neg_one_is_non_residue_of hF]
```

Having the above results at hand, we can show that χ turns out to be "symmetric" or "skew-symmetric" in certain cases:

Lemma 4.3.5.

(1) If $q \equiv 1 \pmod 4$, $\chi(-i) = \chi(i)$ for any $i \in F$. Another form is $\chi(j-i) = \chi(i-j)$ for any $i,\ j \in F$.

(2) If $q \equiv 3 \pmod 4$, $\chi(-i) = -\chi(i)$ for any $i \in F$. Another form is $\chi(j-i) = -\chi(i-j)$ for any $i,\ j \in F$.

Proof.

(1) If $q \equiv 1 \pmod 4$ and $i \in F$,

$$\chi(-i) = 1 * \chi(-i) = \chi(-1) * \chi(-i) = \chi((-1) * (-i)) = \chi(i),$$

where the second equality is given by Lemma 4.3.4, and the third is given by Lemma 4.3.3.

(2) If $q \equiv 3 \pmod 4$ and $i \in F$,

$$-\chi(i) = (-1) * \chi(i) = \chi(-1) * \chi(i) = \chi(-i),$$

where the second equality is given by Lemma 4.3.4, and the third is given by Lemma 4.3.3.

```
/-- `χ (-i) = χ i` if `q ≡ 1 [MOD 4]`. -/
theorem quad_char_is_sym_of (hF : q ≡ 1 [MOD 4]) (i : F) :
χ (-i) = χ i :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- derive the char p of F
  resetI, -- resets the instance cache
  have hp := char_ne_two_of p (or.inl hF), -- hp: p ≠ 2
  have h := char_neg_one_eq_one_of hF, -- h: χ (-1) = 1
  -- χ (-i) = 1 * χ (-i) = χ (-1) * χ (-i) = χ ((-1) * (-i))
  rw [← one_mul (χ (-i)), ← h, ← quad_char_mul hp],
  simp, assumption
end

/-- another form of `quad_char_is_sym_of` -/
theorem quad_char_is_sym_of' (hF : q ≡ 1 [MOD 4]) (i j : F) :
χ (j - i) = χ (i - j) :=
by convert quad_char_is_sym_of hF (i - j); ring

/-- `χ (-i) = - χ i` if `q ≡ 3 [MOD 4]`. -/
theorem quad_char_is_skewsym_of (hF : q ≡ 3 [MOD 4]) (i : F) :
χ (-i) = - χ i :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- derive the char p of F
  resetI, -- resets the instance cache
  have hp := char_ne_two_of p (or.inr hF), -- hp: p ≠ 2
  have h := char_neg_one_eq_neg_one_of hF, -- h: χ (-1) = 1
  rw [← neg_one_mul (χ i), ← h, ← quad_char_mul hp],
  simp, assumption
end

/-- another form of `quad_char_is_skewsym_of` -/
theorem quad_char_is_skewsym_of' (hF : q ≡ 3 [MOD 4]) (i j : F) :
χ (j - i) = - χ (i - j) :=
by convert quad_char_is_skewsym_of hF (i - j); ring
```

■

The tactic `resetI` in the Lean proof resets the instance cache, and thus allows any new instances added to the context to be used in typeclass inference.

We end this section with two results about summations of quadratic characters.

LEMMA 4.3.6. *If $p \neq 2$, we have $\sum_{a \in F} \chi(a) = 0$.*

PROOF.

$$\begin{aligned}
\sum_{a \in F} \chi(a) &= \chi(0) + \sum_{a \neq 0} \chi(a) \\
&= 0 + \sum_{a \in R} \chi(a) + \sum_{a \in N} \chi(a) \\
&= \sum_{a \in R} \chi(a) + \sum_{a \in N} \chi(a) \\
&= \sum_{a \in R} 1 + \sum_{a \in N} (-1) \\
&= |R| - |N| \\
&= 0 \quad \text{by Lemma 4.2.5}
\end{aligned}$$

In the implementation, `quad_char.sum_in_non_zeros_eq_zero` proves $\sum_{a \neq 0} \chi(a) = 0$, and `quad_char.sum_eq_zero` encapsulates it.

```
variable (F) -- use `F` as an explicit parameter
/-- `∑ a : {a : F // a ≠ 0}, χ (a : F) = 0` if `p ≠ 2`. -/
lemma quad_char.sum_in_non_zeros_eq_zero (hp : p ≠ 2):
∑ a : {a : F // a ≠ 0}, χ (a : F) = 0 :=
begin
  simp [fintype.sum_split' (λ a : F, a ≠ 0) (λ a : F, ∃ b, a = b^2)],
  suffices h :
  ∑ (j : {j // j ≠ 0 ∧ ∃ (b : F), j = b ^ 2}), χ (j : F) =
  ∑ a : {a : F // is_quad_residue a} , 1,
  suffices g :
  ∑ (j : {j // j ≠ 0 ∧ ¬∃ (b : F), j = b ^ 2}), χ (j : F) =
  ∑ a : {a : F // is_non_residue a} , -1,
  simp [h, g, sum_neg_distrib,
        card_residues_eq_card_non_residues' F hp],
  any_goals
  {apply fintype.sum_congr, intros a, have := a.2, simp* at *},
end

/-- `∑ (a : F), χ a = 0` if `p ≠ 2`. -/
theorem quad_char.sum_eq_zero (hp : p ≠ 2):
∑ (a : F), χ a = 0 :=
by simp [fintype.sum_split (λ b, b = (0 : F)),
         quad_char.sum_in_non_zeros_eq_zero F hp, default

variable {F} -- use `F` as an implicit parameter
/-- another form of `quad_char.sum_eq_zero` -/
@[simp] lemma quad_char.sum_eq_zero_reindex_1 (hp : p ≠ 2) {a : F}:
∑ (b : F), χ (a - b) = 0 :=
begin
  rw ← quad_char.sum_eq_zero F hp,
  refine fintype.sum_equiv ((equiv.sub_right a).trans (equiv.neg _)) _ _ (by simp),
end
/-- another form of `quad_char.sum_eq_zero` -/
```

```
@[simp] lemma quad_char.sum_eq_zero_reindex_2 (hp : p ≠ 2) {b : F}:
∑ (a : F), χ (a - b) = 0 :=
begin
  rw ← quad_char.sum_eq_zero F hp,
  refine fintype.sum_equiv (equiv.sub_right b) _ _ (by simp),
end
```

■

Lemma 4.3.7. *Suppose $p \neq 2$.*
For $b \in F$ with $b \neq 0$, we have $\sum_{a \in F} \chi(a) * \chi(a+b) = -1$.

An alternative form is, for $b,\ c \in F$ with $b \neq c$, $\sum_{a \in F} \chi(b-a) * \chi(c-a) = -1$.

Proof. Suppose $p \neq 2$ and $0 \neq b \in F$.

$$\begin{aligned}
\sum_{a \in F} \chi(a)\chi(a+b) &= \chi(0)\chi(0+b) + \sum_{a \neq 0} \chi(a)\chi(a+b) \\
&= 0 + \sum_{a \neq 0} \chi(a^{-1})\chi(a+b) \quad \text{by Lemma 4.3.2} \\
&= \sum_{a \neq 0} \chi(a^{-1}(a+b)) \quad \text{by Lemma 4.3.3} \\
&= \sum_{a \neq 0} \chi(1+a^{-1}b) \\
&= \sum_{c \neq 1} \chi(c) \quad \text{by replacing } 1+a^{-1}b \text{ with } c \\
&= \sum_{c \in F} \chi(c) - \chi(1) \\
&= 0 - 1 \quad \text{by Lemma 4.3.6} \\
&= -1
\end{aligned}$$

```
variable {F} -- use `F` as an implicit parameter
/-- helper of `quad_char.sum_mul'`: reindex the terms in the summation -/
lemma quad_char.sum_mul'_aux {b : F} (hb : b ≠ 0) :
∑ (a : F) in filter (λ (a : F), ¬a = 0) univ, χ (1 + a⁻¹ * b) =
∑ (c : F) in filter (λ (c : F), ¬c = 1) univ, χ (c) :=
begin
  refine finset.sum_bij (λ a ha, 1 + a⁻¹ * b) (λ a ha, _)
    (λ a ha, rfl) (λ a₁ a₂ h1 h2 h, _) (λ c hc, _),
  { simp at ha, simp* },
  { simp at h1 h2, field_simp at h, rw (mul_right_inj' hb).1 h.symm },
  { simp at hc, use b * (c - 1)⁻¹,
    simp [*, mul_inv_rev', sub_ne_zero.2 hc] }
end

/-- If `b ≠ 0` and `p ≠ 2`, `∑ a : F, χ (a) * χ (a + b) = -1`. -/
theorem quad_char.sum_mul' {b : F} (hb : b ≠ 0) (hp : p ≠ 2):
∑ a : F, χ (a) * χ (a + b) = -1 :=
begin
  rw [finset.sum_split _ (λ a, a = (0 : F))],
```

```
  simp,
  have h : ∑ (a : F) in filter (λ (a : F), ¬a = 0) univ, χ a * χ (a + b) =
           ∑ (a : F) in filter (λ (a : F), ¬a = 0) univ, χ (1 + a⁻¹ * b),
  { apply finset.sum_congr rfl,
    intros a ha, simp at ha,
    simp [←quad_char_inv a, ←quad_char_mul hp a⁻¹],
    field_simp },
  rw [h, quad_char.sum_mul'_aux hb],
  have g:= @finset.sum_split _ _ _ (@finset.univ F _) (χ) (λ a : F, a = 1) _,
  simp [quad_char.sum_eq_zero F hp] at g,
  linarith
end

/-- another form of `quad_char.sum_mul'` -/
theorem quad_char.sum_mul {b c : F} (hbc : b ≠ c) (hp : p ≠ 2):
∑ a : F, χ (b - a) * χ (c - a) = -1 :=
begin
  rw ← quad_char.sum_mul' (sub_ne_zero.2 (ne.symm hbc)) hp,
  refine fintype.sum_equiv ((equiv.sub_right b).trans (equiv.neg _)) _ _ (by simp),
end
```

■

5 HADAMARD MATRICES

This section defines Hadamard matrices and proves some basic results about Hadamard matrices.

- Throughout this chapter, `H` denotes a square matrix and in most parts has type `matrix I I ℚ`. In the context outside the codes, $n \in \mathbb{Q}$ denotes the cardinality of `I`.
- The implementation in this chapter mainly follows [20] and also uses [11] as a reference.
- Please see file MAIN2.LEAN for the complete code work of this chapter's topic.

5.1 Basic Definitions

DEFINITION 5.1.1. *A **Hadamard matrix**[1] H is a square matrix whose entries are either* 1 *or* -1 *and whose rows are pairwise orthogonal.*

Remark. In particular, for an $n \times n$ Hadamard matrix H, one has $H \cdot H^T = nI$, implying that $|\det H| = n^{n/2}$, the maximal possible value for the absolute value of the determinant of a matrix with absolute values of entries bounded by 1.

The Hadamard matrices is formalised as the below type class, which itself is a proposition:

```
class Hadamard_matrix (H : matrix I I ℚ) : Prop :=
(one_or_neg_one []: ∀ i j, (H i j) = 1 ∨ (H i j) = -1
(orthogonal_rows []: H.has_orthogonal_rows)
```

Thus, to prove a matrix H is a Hadamard matrix is to construct an instance of the type class `Hadamard_matrix H`. `Hadamard_matrix H` contains two fields: `one_or_neg_one`, a proof that the entries of H are either 1 or -1, and `orthogonal_rows`, a poof that H has mutually orthogonal rows. Therefore, with the use of the anonymous constructor, `⟨p, q⟩`, where `p` is a proof of `∀ i j, (H i j) = 1 ∨ (H i j) = -1` and `q` is a proof of `H.has_orthogonal_rows`, is an instance of `Hadamard_matrix H`. The linked proofs are simple such examples.

Remark 1. The square brackets after `one_or_neg_one` makes `(H : matrix I I ℚ)` an explicit argument to `one_or_neg_one`, and so does the other pair.

Hence, `# check Hadamard_matrix.one_or_neg_one` prints

```
Hadamard_matrix.one_or_neg_one :
∀ (H : matrix ?M_1 ?M_1 ℚ) [c : H.Hadamard_matrix] (i j : ?M_1),
H i j = 1 ∨ H i j = -1
```

where `?M_1` is meta variable, denoting an undetermined type.

Remark 2. The reason that the entries of the parametric matrix H have been chosen to be in $\mathbb{Q}$ is for convenience, as $\mathbb{Q}$ is the smallest field containing -1 and 1. If we use $\mathbb{Z}$ instead, we have to coerce many of the numbers in $\mathbb{Z}$ into $\mathbb{Q}$ in the following implementation; if we use $\mathbb{R}$, some of the following lemmas and constructions can only be `noncompuatable` if not putting much more efforts. Indeed, we can let S denote the subtype of $\mathbb{Q}$ which only contains 1 and -1 and make an alternative definition `Hadamard_matrix'` of `Hadamard_matrix`, which takes parameter `(H : matrix I I S)`, but again, this definition requires us to build and use coercion from S to $\mathbb{Q}$:

```
private abbreviation S := {x : ℚ// x = 1 ∨ x = -1}
instance fun_S_to_ℚ: has_coe (β → S) (β → ℚ) := ⟨λ f x, f x⟩
class Hadamard_matrix' (H : matrix I I S):=
(orthogonal_rows []: ∀ i₁ i₂, i₁ ≠ i₂ → dot_product ((H i₁) : (I → ℚ)) (H i₂) = 0)
```

Throughout, we will use the definition `Hadamard_matrix`.

We make two more auxiliary definitions:

[1] Named after the French mathematician Jacques Hadamard.

Definition 5.1.2. *Give a matrix H and any rows i_1, i_2 of H, we define:*

- *matched H i_1 i_2 is the set of j's such that $H\ i_1\ j = H\ i_2\ j$*
- *mismatched H i_1 i_2 is the set of j's such that $H\ i_1\ j \neq H\ i_2\ j$*

For our purpose, we implement H to have type `matrix I I ℚ`:

```
@[reducible, simp]
def matched (H : matrix I I ℚ) (i₁ i₂ : I) : set I :=
{j : I | H i₁ j = H i₂ j}
@[reducible, simp]
def mismatched (H : matrix I I ℚ) (i₁ i₂ : I) : set I :=
{j : I | H i₁ j ≠ H i₂ j}
```

We implemented `matched H i₁ i₂` and `mismatched H i₁ i₂` to have type `set I`. `set α` in mathlib is implemented as the type of one-ary predicates.

```
def set (α : Type u) := α → Prop
```

That is, a set S of elements in `α` is implemented to have type `set α` and is by definition a one-ary predicate on `α`. Mathlib use the usual notation `{a : α | S a}` to denote a set `S`. As the two definitions above serve as abbreviations, we add the tag `reducible` so that Lean's elaborator unfolds the definitions eagerly [2, sec. 6.10]. The `simp` tag makes the `simp` tactic unfold the definitions automatically.

We establish some basic results/APIs for *matched H* and *mismatched H* without the assumption that H is a Hadamard matrix:

```
/-- `matched H i₁ i₂ ∪ mismatched H i₁ i₂ = I` as sets -/
@[simp] lemma match_union_mismatch (H : matrix I I ℚ) (i₁ i₂ : I) :
matched H i₁ i₂ ∪ mismatched H i₁ i₂ = @set.univ I

/-- `matched H i₁ i₂` and `mismatched H i₁ i₂` are disjoint as sets -/
@[simp] lemma disjoint_match_mismatch (H : matrix I I ℚ) (i₁ i₂ : I) :
disjoint (matched H i₁ i₂) (mismatched H i₁ i)

/-- `|I| = |H.matched i₁ i₂| + |H.mismatched i₁ i₂|`
    for any rows `i₁` `i₂` of a matrix `H` with index type `I`-/
lemma card_match_add_card_mismatch [decidable_eq I] (H : matrix I I ℚ) (i₁ i₂ : I) :
set.card (@set.univ I) = set.card (matched H i₁ i₂) + set.card (mismatched H i₁ i₂)

lemma dot_product_split [decidable_eq I] (H : matrix I I ℚ) (i₁ i₂ : I) :
∑ j in (@set.univ I).to_finset, H i₁ j * H i₂ j =
∑ j in (matched H i₁ i₂).to_finset, H i₁ j * H i₂ j +
∑ j in (mismatched H i₁ i₂).to_finset, H i₁ j * H i₂ j
```

In the sigma-sum `∑ x in s, f` implemented in mathlib, `s` is required to be a finite set or a finite type. The function `set.to_finset` in mathlib converts a set to a finite set (more precisely, converts an object of type `set α` for some `α` to an object of type `finset α`) when it is possible to do so.

5.2 Basic properties

In this section, we establish some basic results of Hadamard matrices. This is the Lean environment setting for this section:

```
namespace Hadamard_matrix
variables (H : matrix I I ℚ) [Hadamard_matrix H]
```

We first build some APIs that will help us to prove more involved lemma/theorems:

```
lemma entry_eq_one_of_ne_neg_one {i j : I} (h : H i j ≠ -1) : H i j = 1

lemma entry_eq_neg_one_of_ne_one {i j : I} (h : H i j ≠ 1) : H i j = -1

lemma entry_eq_neg_one_of
{i j k l : I} (h : H i j ≠ H k l) (h' : H i j = 1) : H k l = -1

lemma entry_eq_one_of
{i j k l : I} (h : H i j ≠ H k l) (h' : H i j = -1) : H k l = 1

lemma entry_eq_entry_of
{a b c d e f : I} (h₁: H a b ≠ H c d) (h₂: H a b ≠ H e f) : H c d = H e f

@[simp] lemma entry_mul_of_ne {i j k l : I} (h : H i j ≠ H k l):
(H i j) * (H k l) = -1

/-- The dot product of a row with itself equals card I. -/
@[simp] lemma row_dot_product_self (i : I) :
dot_product (H i) (H i) = card I

/-- The dot product of a column with itself equals card I. -/
@[simp] lemma col_dot_product_self (j : I) :
dot_product (λ i, H i j) (λ i, H i j) = card I

/-- The dot product of a row with another row equals 0. -/
@[simp] lemma row_dot_product_other {i₁ i₂ : I} (h : i₁ ≠ i₂) :
dot_product (H i₁) (H i₂) = 0

/- card_match_eq -/
lemma card_match_eq {i₁ i₂ : I} (h: i₁ ≠ i₂):
(set.card (matched H i₁ i₂) : ℚ) =
∑ j in (matched H i₁ i₂).to_finset, H i₁ j * H i₂ j

/- card_mismatch_eq -/
lemma card_mismatch_eq {i₁ i₂ : I} (h: i₁ ≠ i₂):
(set.card (mismatched H i₁ i₂) : ℚ) =
- ∑ j in (mismatched H i₁ i₂).to_finset, H i₁ j * H i₂ j
```

The names and statements of such APIs are self-explaining: e.g. the lemma `entry_eq_one_of_ne_neg_one` proves that if `H i j ≠ -1`, then `H i j = 1`.

Lemma 5.2.1. $H \cdot H^T = nI$.

Proof. Every non-diagonal entry of $H \cdot H^T$, which is the dot product of two distinct rows of H, equals zero, as the rows of H are pairwise orthogonal; every diagonal entry of $H \cdot H^T$, which is the dot product of a row of H with itself, equals n, as each row of H is a vector of entries 1 or -1.

```
lemma mul_tanspose [decidable_eq I] :
H · Hᵀ = (card I : ℚ) • 1 :=
begin
  ext,
  simp [transpose, matrix.mul],
  by_cases i = j; simp [*, mul_one] at *,
end
```

■

This immediately follows that

LEMMA 5.2.2. $H \cdot \frac{1}{n} H^T = 1$

PROOF.
if $n = 0$, both sides of the equality are empty matrices, and hence are equal;
if $n > 0$, as $\frac{1}{n} * n = 1$, the simplifier is able to deduce the equation from Lemma 5.2.1.
Note: in Lean, as $\frac{1}{n}$ is defined to be 0 if $n = 0$;thus, the two cases can not be combined.

```
lemma right_invertible [decidable_eq I] :
  H · ((1 / (card I : ℚ)) • Hᵀ) = 1 :=
begin
  have h := mul_tanspose H,
  by_cases hI : card I = 0,
  {exact @eq_of_empty _ _ _ (card_eq_zero_iff.mp hI) _ _}, -- the trivial case
  have hI': (card I : ℚ) ≠ 0, {simp [hI]},
  simp [h, hI'],
end
```

■

As $\frac{1}{n} H^T$ is the right inverse of H, `invertible_of_right_inverse` deduces that H is invertible, and `inv_eq_right_inv` proves that $\frac{1}{n} H^T$ is the inverse. `invertible_of_right_inverse` and `inv_eq_right_inv` were missing in mathlib. I contributed various lemmas including these two about matrix inverse into mathlib (please see INV_MATRIX.LEAN for the full contribution).

LEMMA 5.2.3. $H^T \cdot H = nI$

PROOF. As $\frac{1}{n} H^T$ is the right inverse of H, it is also the left inverse (i.e. $\frac{1}{n} H^T \cdot H = 1$). Again, a by-case discussion of if $n = 0$ in Lean gives the conclusion.

```
lemma right_invertible [decidable_eq I] :
H · ((1 / (card I : ℚ)) • Hᵀ) = 1 :=
begin
have h := mul_tanspose H,
by_cases hI : card I = 0,
{exact @eq_of_empty _ _ _ (card_eq_zero_iff.mp hI) _ _}, -- the trivial case
have hI': (card I : ℚ) ≠ 0, {simp [hI]},
simp [h, hI'],
end
```

■

Lemma 5.2.3 implies that a Hadamard matrix also has orthogonal columns and other basic results/APIs in the next code block:

LEMMA 5.2.4. *A Hadamard matrix H has mutually orthogonal columns.*

```
/-- The dot product of a column with another column equals `0`. -/
@[simp] lemma col_dot_product_other [decidable_eq I] {j₁ j₂ : I} (h : j₁ ≠ j₂) :
dot_product (λ i, H i j₁) (λ i, H i j₂) = 0 :=
begin
  have h':= congr_fun (congr_fun (tanspose_mul H) j₁) j₂,
  simp [matrix.mul, transpose, has_one.one, diagonal, h] at h',
  assumption,
end

/-- Hadamard matrix `H` has orthogonal rows-/
@[simp] lemma has_orthogonal_cols [decidable_eq I] :
```

```
H.has_orthogonal_cols:=
by intros i j h; simp [h]

/-- `Hᵀ` is a Hadamard matrix suppose `H` is. -/
instance transpose [decidable_eq I] : Hadamard_matrix Hᵀ :=
begin
  refine{..},
  -- first goal
  {intros, simp[transpose]},
  -- second goal
  simp [transpose_has_orthogonal_rows_iff_has_orthogonal_cols]
end

/-- `Hᵀ` is a Hadamard matrix implies `H` is a Hadamard matrix.-/
lemma of_Hadamard_matrix_transpose [decidable_eq I]
{H : matrix I I ℚ} (h: Hadamard_matrix Hᵀ):
Hadamard_matrix H :=
by convert Hadamard_matrix.transpose Hᵀ; simp
```

The `refine` tactic used above in the proof of `transpose` splits the construction of an element of some structure into separate goals. In our situation, it splits into two goals: to prove the matrix in question has entries as 1 or -1, and to prove the matrix has orthogonal rows.

Recall that the cardinality function is only implemented for finite sets. I implemented `set.card` as a shortcut for `finset.card set.to_finset` and relevant properties of `set.card` in SET_FINSET_FINTYPE.LEAN.

LEMMA 5.2.5. *For any distinct rows i_1,i_2 we have $|matched\ H\ i_1\ i_2| = |mismatched\ H\ i_1\ i_2|$*

PROOF. Let A = match $H\ i_1\ i_2$ and B = mismatched $H\ i_1\ i_2$. Then,

$$\begin{aligned} 0 &= (H\ i_1) \cdot (H\ i_2) \text{ as distinct rows of } H \text{ are orthogonal} \\ &= \sum_{j \in A} (H\ i_1\ j)(H\ i_2\ j) + \sum_{j \in B} (H\ i_1\ j)(H\ i_2\ j) \\ &= \sum_{j \in A} 1 + \sum_{j \in B} (-1) \\ &= |A| - |B| \end{aligned}$$

Thus, $|A| = |B|$.
In the Lean implementation, `card_match_eq` proves $\sum_{j \in A}(H\ i_1\ j)(H\ i_2\ j) = |A|$, and `card_mismatch_eq` proves $\sum_{j \in B}(H\ i_1\ j)(H\ i_2\ j) = -|B|$.

```
lemma card_match_eq_card_mismatch_ℚ
[decidable_eq I] {i₁ i₂ : I} (h: i₁ ≠ i₂):
(set.card (matched H i₁ i₂) : ℚ)= set.card (mismatched H i₁ i₂) :=
begin
  have eq := dot_product_split H i₁ i₂,
  rw [card_match_eq H h, card_mismatch_eq H h],
  simp only [set.to_finset_univ, row_dot_product'_other H h] at eq,
  linarith,
end
```

■

Another useful API is

```
lemma reindex (f : I ≃ J) (g : I ≃ J): Hadamard_matrix (reindex f g H)
```

which proves that if H is a Hadamard matrix, then any matrix produced by reindexing the columns and rows of H is also a Hadamard matrix. `reindex f g H` is the reindexed matrix given the row-reindex map `f`, and the column-reindex map `g`. A coming example, in Chapter 6 about the Sylvester's constructions, will demonstrate why this API can be useful.

5.3 Basic constructions

This section constructs Hadamard matrices of order 0, 1, 2.

0. The empty square matrix is implemented as:

```
def H_0 : matrix empty empty ℚ := 1
```

1. The order 1 matrix [1] is implemented as two versions:

```
def H_1 : matrix unit unit ℚ := 1

def H_1' : matrix punit punit ℚ := λ i j, 1
```

The difference is merely concerning types: `unit` has type `Type`; `punit` is the universe polymorphic version of `unit`, and has type `Sort u` (= `Type u - 1`). The second one turns out to be useful for proving Theorem 6.1.4.

2. The order 2 matrix $\left[\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right]$ is implemented as:

```
def H_2 : matrix (unit ⊕ unit) (unit ⊕ unit) ℚ :=
(1 :matrix unit unit ℚ).from_blocks 1 1 (-1)
```

`A.from_blocks B C D` is the block matrix $\left[\begin{smallmatrix} A & B \\ C & D \end{smallmatrix}\right]$.

We now implement the proofs that those matrices are all Hadamard matrices:

```
instance Hadamard_matrix.H_0 : Hadamard_matrix H_0 := ⟨by tidy, by tidy⟩

instance Hadamard_matrix.H_1 : Hadamard_matrix H_1 := ⟨by tidy, by tidy⟩

instance Hadamard_matrix.H_1' : Hadamard_matrix H_1' := ⟨by tidy, by tidy⟩

instance Hadamard_matrix.H_2 : Hadamard_matrix H_2 :=
⟨ by tidy,
  λ i₁ i₂ h,
    by { cases i₁, any_goals {cases i₂},
         any_goals {simp[*, H_2, dot_product, fintype.sum_sum_type] at *}, }
⟩
```

Again, the `instance` command declares the lemmas to be instances of certain typeclasses. As the poofs are all simple, most of the goals can be closed by the tactic `tidy`, which tries a variety of conservative tactics to solve goals.

5.4 Normalisation

In this section, we set variables:

```
variables (H : matrix I I ℚ) [Hadamard_matrix H]
```

Definition 5.4.1. *A Hadamard matrix is called* ***normalised*** *or in* ***normal form*** *if all the elements in its first row and first column are* +1.

In Lean, the type `I` does not necessarily have a notion of "the first" or an element called 1. To make the implementation of Definition 5.4.1 as general as possible, we merely require the condition `[inhabited I]`, which means `I` has a distinguished element `default I`.

```
/-- normalised Hadamard matrix -/
def is_normalized [inhabited I] : Prop :=
H (default I) = 1 ∧ (λ i, H i (default I)) = 1
```

The claim that every Hadamard matrix can be normalised comes from Lemmas 5.4.2 and 5.4.3. When doing a proof by hand or presenting a proof on paper, there might be advantages to assume a given Hadamard matrix is in normal form when possible; while when implementing a theorem in Lean, as a computer proof assistant, assuming a given Hadamard matrix is in normal form usually does not give us any advantage but restricts the assumption.

To implement Lemmas 5.4.2 and 5.4.3 we define two auxiliary definitions. In the following content, negating a row/column of matrix means replace every entry a of this row/column by its negation $-a$. As H is over $\mathbb{Q}$, we may also refer negating a row/column of H as multiplying a row/column of H by -1.

```
/-- negate row `i` of matrix `A`; `[decidable_eq I]` is required by `update_row` -/
def neg_row [has_neg α] [decidable_eq I] (A : matrix I J α) (i : I) :=
update_row A i (- A i)

/-- negate column `j` of matrix `A`; `[decidable_eq J]` is required by `update_column` -/
def neg_col [has_neg α] [decidable_eq J] (A : matrix I J α) (j : J) :=
update_column A j (-λ i, A i j)
```

A rule of thumb of writing mathlib is every new definition should be followed by APIs for end users. The following APIs are enough for the purpose of this project:

```
/-- Negating row `i` and then column `j` equals negating column `j` first
    and then row `i`. -/
lemma neg_row_neg_col_comm [has_neg α] [decidable_eq I] [decidable_eq J]
  (A : matrix I J α) (i : I) (j : J) :
  (A.neg_row i).neg_col j = (A.neg_col j).neg_row i

lemma transpose_neg_row [has_neg α] [decidable_eq I] (A : matrix I J α) (i : I) :
  (A.neg_row i)ᵀ = Aᵀ.neg_col i :=
  by simp [← update_column_transpose, neg_row, neg_col]

lemma transpose_neg_col [has_neg α] [decidable_eq J] (A : matrix I J α) (j : J) :
  (A.neg_col j)ᵀ = Aᵀ.neg_row j :=
  by {simp [← update_row_transpose, neg_row, neg_col, trans_row_eq_col]}

lemma neg_row_add [add_comm_group α] [decidable_eq I]
  (A B : matrix I J α) (i : I) :
  (A.neg_row i) + (B.neg_row i) = (A + B).neg_row i

lemma neg_col_add [add_comm_group α] [decidable_eq J]
  (A B : matrix I J α) (j : J) :
  (A.neg_col j) + (B.neg_col j) = (A + B).neg_col j

/-- Negating the same row and column of diagonal matrix `A` equals `A` itself. -/
lemma neg_row_neg_col_eq_self_of_is_diag [add_group α] [decidable_eq I]
  {A : matrix I I α} (h : A.is_diagonal) (i : I) :
  (A.neg_row i).neg_col i = A
```

Lemma 5.4.2. *Multiplying any row by -1 changes a Hadamard matrix H into another Hadamard matrix.*

```
/-- Negating any row `i` of a Hadamard matrix `H` produces another Hadamard matrix. -/
instance Hadamard_matrix.neg_row (i : I) :
Hadamard_matrix (H.neg_row i) :=
begin
  refine {..},
  -- first goal
  { intros j k,
    simp [neg_row,  update_row_apply],
    by_cases j = i; simp* at * },
  -- second goal
  { intros j k hjk,
    -- a by cases discussion of if the rows `j` `k` equal the negated row `i`.
    by_cases h1 : j = i, any_goals {by_cases h2 : k = i},
    -- simplify all the cases
    any_goals {simp [*, neg_row, update_row_apply]},
    tidy }
end
```

Lemma 5.4.3. *Multiplying any column by -1 changes a Hadamard matrix H into another Hadamard matrix.*

```
/-- Negating any column `j` of a Hadamard matrix `H` produces another Hadamard matrix. -/
instance Hadamard_matrix.neg_col (j : I) :
Hadamard_matrix (H.neg_col j) :=
begin
  apply of_Hadamard_matrix_transpose, -- changes goal to `(H.neg_col j)ᵀ.Hadamard_matrix`
  simp [transpose_neg_col, Hadamard_matrix.neg_row]
  -- `(H.neg_col j)ᵀ = Hᵀ.neg_row j`, where the RHS has been proved to be a Hadamard matrix.
end
```

`Hadamard_matrix.neg_row` uses "proving by definition" via the `refine` tactic; `Hadamard_matrix.neg_col` takes the advantage of some established results, such as `Hadamard_matrix.neg_row`, and is thus a nicer and shorter implementation.

5.5 Special type Hadamard matrices

In this section, we define two special types of Hadamard matrices. We set variables:

```
variables (H : matrix I I ℚ) [Hadamard_matrix H]
```

Definition 5.5.1. *A Hadamard matrix H is **regular** if its row and column sums are all equal.*

```
/-- regular Hadamard matrix -/
def is_regular : Prop :=
∀ i j, H.row_sum i = H.col_sum j
```

Definition 5.5.2. *A Hadamard matrix H is **skew** if $H^T + H = 2I$.*

```
/-- skew Hadamard matrix -/
def is_skew [decidable_eq I] : Prop :=
Hᵀ + H = 2
```

Lemma 5.5.3. *A skew Hadamard matrix remains a skew Hadamard matrix after multiplication of any row and its corresponding column by -1.*

Proof. Suppose H is a Hadamard matrix with index set I, and i is in I. Let f be the function that negating row i of a given matrix, and g be the function that negating column i of a given matrix.
By Lemmas 5.4.2 and 5.4.3, $g(f(H))$ is a Hadamard matrix.
To show $g\ f\ H$ is skew is to show $(g\ f\ H)^T + g\ f\ H = 2I$.

$$\begin{aligned}(g\ f\ H)^T + g\ f\ H &= (f\ g\ H)^T + g\ f\ H \\ &= g\ f\ (H^T) + g\ f\ H \\ &= g\ f\ (H^T + H) \\ &= g\ f\ (2I) \ \text{ as } H \text{ is skew} \\ &= 2I\end{aligned}$$

```
lemma is_skew.of_neg_one_smul_row_col_of_is_skew
[decidable_eq I] (i : I) (h : Hadamard_matrix.is_skew H) :
is_skew ((H.neg_row i).neg_col i) :=
begin
  simp [is_skew],
  nth_rewrite 0 [neg_row_neg_col_comm],
  simp [transpose_neg_row, transpose_neg_col, neg_row_add, neg_col_add],
  rw [h.eq],
  convert neg_row_neg_col_eq_self_of_is_diag _ _,
  apply is_diagonal_add; by simp
end
```

■

6 SYLVESTER'S CONSTRUCTIONS

This section implements Sylvester's constructions of Hadamard matrices including two versions of Sylvester's original construction `Sylvester_constr₀` and `Sylvester_constr₀'`, and the generalized Sylvester's construction `Sylvester_constr`.

- Throughout this chapter, `H` denotes a square matrix and in most parts has type `matrix I I ℚ`. In the context outside the codes, $n \in \mathbb{Q}$ denotes the cardinality of `I`.
- The implementation in this chapter mainly follows [20] and also uses [11] as a reference.
- Please see file MAIN2.LEAN for the complete code work of this chapter's topic.

6.1 Sylvester's construction

DEFINITION 6.1.1. *Let H be a Hadamard matrix of order n, the **Sylvester's construction** is the matrix $\begin{bmatrix} H & H \\ H & -H \end{bmatrix}$ of order $2n$.*
We can have two implementations of this definition:

```
-- Version 1
def Sylvester_constr₀ (H : matrix I I ℚ) [Hadamard_matrix H] :
matrix (I ⊕ I) (I ⊕ I) ℚ :=
H.from_blocks H H (-H)

-- Version 2
def Sylvester_constr₀' (H : matrix I I ℚ) [Hadamard_matrix H]:
matrix (I × (unit ⊕ unit)) (I × (unit ⊕ unit)) ℚ :=
H ⊗ H_2
```

The difference is that the first one is constructed via the function `matrix.from_blocks` for constructing block matrices and the second one is constructed via Kronecker product with `H_2` (recall $H_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$). Thus, the resulting constructions have different types:

- Version 1 is of type `matrix (I ⊕ I) (I ⊕ I) ℚ`,
- Version 2 is of type `matrix (I × (unit ⊕ unit)) (I × (unit ⊕ unit)) ℚ`.

We will first proof that Version 1 is a Hadamard matrix and then show Version 2 is a Hadamard matrix as well by a different trick.

THEOREM 6.1.2. *The Sylvester's construction, implemented as `Sylvester_constr₀`, is a Hadamard matrix.*

PROOF. Let $S = \begin{bmatrix} H & H \\ H & -H \end{bmatrix}$, where H is a Hadamard matrix with index set $I_1 \oplus I_2$ (we may not distinguish sets and types in the proofs in words). Obviously for each $i, j \in I_1 \oplus I_2$ we have $S\ i\ j \in \{1, -1\}$.
For $i \neq j \in I_1 \oplus I_2$,

(1) If $i, j \in I_1$ then $S\ i \cdot S\ j = H\ i \cdot H\ j + H\ i \cdot H\ j = 0 + 0 = 0$

Similarly,

(2) If $i \in I_1$ and $j \in I_2$, then $S\ i \cdot S\ j = H\ i \cdot H\ j - H\ i \cdot H\ j = 0$.
(3) If $i, j \in I_2$, then $S\ i \cdot S\ j = H\ i \cdot H\ j + H\ i \cdot H\ j = 0$.

```
@[instance] theorem Hadamard_matrix.Sylvester_constr₀
(H : matrix I I ℚ) [Hadamard_matrix H] :
Hadamard_matrix (matrix.Sylvester_constr₀ H) :=
begin
  -- the first goal
  refine{..},
  { rintros (i | i)  (j | j);
    simp [matrix.Sylvester_constr₀] },
  -- the second goal
```

```
  rintros (i | i) (j | j) h,
  all_goals {simp [matrix.Sylvester_constr₀, dot_product_block', *]},
  any_goals {rw [← dot_product], have h' : i ≠ j; simp* at *}
end
```

■

When defining the two versions of implementation, a sharp reader might have already noticed that there is a natural map from the index set `I ⊕ I` to `I × (unit ⊕ unit)`. We implement the "reindex" map and the proof that the two versions are equal up to reindexing.

```
-- the "reindex" map
def sum_self_equiv_prod_unit_sum_unit : I ⊕ I ≃  I × (unit ⊕ unit)

-- an abbreviation of the "reindex" map
local notation `reindex_map` := equiv.sum_self_equiv_prod_unit_sum_unit

-- a proof that the second version equals the first version after being reindexed
lemma Sylvester_constr₀'_eq_reindex_Sylvester_constr₀
(H : matrix I I ℚ) [Hadamard_matrix H] :
H.Sylvester_constr₀' = reindex reindex_map reindex_map H.Sylvester_constr₀
```

Then with the help of the built API `Hadamard_matrix.reindex`, we can obtain a nice and short proof for Version 2:

THEOREM 6.1.3. *The Sylvester's construction, implemented as* `Sylvester_constr₀'`*, is a Hadamard matrix.*

PROOF. Reindexing the rows and columns of a Hadamard matrix produces another Hadamard matrix.

```
@[instance] theorem Hadamard_matrix.Sylvester_constr₀'
(H : matrix I I ℚ) [Hadamard_matrix H] :
Hadamard_matrix (Sylvester_constr₀' H) :=
begin
  convert Hadamard_matrix.reindex H.Sylvester_constr₀ reindex_map reindex_map,
  exact H.Sylvester_constr₀'_eq_reindex_Sylvester_constr₀,
end
```

■

Remark. Theorem 6.1.3 is in fact not necessary as `Sylvester_constr₀'` turns out to be a special case of the more generalized construction `Sylvester_constr` that will be implemented and proved to be a Hadamard matrix in the next section. The purpose for presenting this proof is to illustrate why the API `Hadamard_matrix.reindex` can be helpful and save efforts in more complicated examples.

With the implementation of the Sylvester's construction, we are now able to establish an order conclusion about Hadamard matrices.

THEOREM 6.1.4. *For any $n \in \mathbb{N}$, there is a Hadamard matrix of order 2^n.*

PROOF. Proceed by induction on n. If $n = 0$ then $[1]$ is a Hadamard matrix with order $2^0 = 1$. Suppose H is a Hadamard matrix with index set I and order 2^n. Then the Sylvester's construction $\left[\begin{smallmatrix} H & H \\ H & -H \end{smallmatrix}\right]$ is a Hadamard matrix with index set $I \oplus I$, and $|I \oplus I| = 2|I| = 2^{n+1}$.

```
theorem Hadamard_matrix.order_conclusion_1:
∀ (n : ℕ), ∃ {I : Type*} [inst : fintype I]
(H : @matrix I I inst inst ℚ) [@Hadamard_matrix I inst H],
@fintype.card I inst = 2^n :=
begin
  intro n,
  induction n with n ih,
  -- the case 0
```

```
  {exact ⟨punit, infer_instance, H_1', infer_instance, by simp⟩},
  -- the case n.succ
  rcases ih with ⟨I, inst, H, h, hI⟩ ,-- unfold the IH
  resetI, -- resets the instance cache
  refine ⟨I ⊕ I, infer_instance, H.Sylvester_constr₀, infer_instance, _⟩,
  rw [fintype.card_sum, hI], ring_nf, -- this line proves `card (I ⊕ I) = 2 ^ n.succ`
end
```

■

6.2 Generalised Sylvester's construction

Definition 6.2.1. *Let H_1 and H_2 be Hadamard matrices of order n, the **generalised Sylvester's construction** is the matrix $H_1 \otimes H_2$.*

The implementation is straight forward with the implemented Kronecker product at hand:

```
def Sylvester_constr
(H₁ : matrix I I ℚ) [Hadamard_matrix H₁] (H₂ : matrix J J ℚ) [Hadamard_matrix H₂] :
matrix (I × J) (I × J) ℚ := H₁ ⊗ H₂
```

theorem 6.2.2. *The generalised Sylvester's construction is a Hadamard matrix.*

Proof. Suppose H_1, H_2 are Hadamard matrices with index sets I, J respectively.

(1) Every entry of $H_1 \otimes H_2$ is of the form $H_1\ i_1\ j_1 * H_2\ i_2\ j_2$.
As $H_1\ i_1\ j_1,\ H_2\ i_2\ j_2 \in \{1, -1\}$, $H_1\ i_1\ j_1 * H_2\ i_2\ j_2 \in \{1, -1\}$.

(2) Suppose $(i_1,\ j_1), (i_2,\ j_2)$ are two distinct row indices of $H_1 \otimes H_2$.
Then

$$\begin{aligned} A &:= H_1 \otimes H_2\ (i_1,\ j_1) * H_1 \otimes H_2\ (i_2,\ j_2) \\ &= \underbrace{(H_1\ i_1 * H_1\ i_2)}_{=:B} * \underbrace{(H_2\ j_1 * H_2\ j_2)}_{=:C} \end{aligned}$$

If $i_1 \neq i_2$, then $B = 0$, and thus $A = 0$.
If $i_1 = i_2$, then $j_1 \neq j_2$, and then $C = 0$, and thus $A = 0$.

```
@[instance] theorem Hadamard_matrix.Sylvester_constr'
(H₁ : matrix I I ℚ) [Hadamard_matrix H₁] (H₂ : matrix J J ℚ) [Hadamard_matrix H₂] :
Hadamard_matrix (H₁ ⊗ H₂) :=
begin
  refine {..},
  -- first goal
  { rintros ⟨i₁, j₁⟩ ⟨i₂, j₂⟩,
    simp [Kronecker],
    -- the current goal : H₁ i₁ i₂ * H₂ j₁ j₂ = 1 ∨ H₁ i₁ i₂ * H₂ j₁ j₂ = -1
    obtain (h | h) := one_or_neg_one H₁ i₁ i₂; -- prove by cases : H₁ i₁ i₂ = 1 or -1
    simp [h] },
  -- second goal
  rintros ⟨i₁, j₁⟩ ⟨i₂, j₂⟩ h,
  simp [dot_product_Kronecker_row_split],
  -- by cases j₁ = j₂; simp* closes the case j₁ ≠ j₂
  by_cases hi: i₁ = i₂, any_goals {simp*},
  -- the left case: i₁ = i₂
  by_cases hi: j₁ = j₂, any_goals {simp* at *},
end

/-- wraps `Hadamard_matrix.Sylvester_constr'`-/
```

```
@[instance] theorem Hadamard_matrix.Sylvester_constr
(H₁ : matrix I I ℚ) [Hadamard_matrix H₁] (H₂ : matrix J J ℚ) [Hadamard_matrix H₂] :
Hadamard_matrix (Sylvester_constr H₁ H₂) :=
Hadamard_matrix.Sylvester_constr' H₁ H₂
```

■

The generalised Sylvester's construction gives us another order conclusion:

THEOREM 6.2.3. *If there is a Hadamard matrix H_1 of order n and a Hadamard matrix H_2 of order m, then there is a Hadamard matrix of $n \times m$.*

PROOF. The generalised Sylvester's construction $H_1 \otimes H_2$ is a Hadamard matrix of order $n \times m$.

The formalisation uses `by exactI` to enable the typeclass inference to use variables in the context, and the implemented proof uses the anonymous construct to provide objects after the "∃".

```
theorem {u v} Hadamard_matrix.order_conclusion_2
{I : Type u} {J : Type v} [fintype I] [fintype J]
(H₁ : matrix I I ℚ) [Hadamard_matrix H₁] (H₂ : matrix J J ℚ) [Hadamard_matrix H₂] :
∃ {K : Type max u v} [inst : fintype K] (H : @matrix K K inst inst ℚ),
by exactI Hadamard_matrix H ∧ card K = card I * card J :=
⟨(I × J), _, Sylvester_constr H₁ H₂, ⟨infer_instance, card_prod I J⟩⟩
```

■

7 PALEY'S CONSTRUCTIONS

This chapter implements Paley construction I `Paley_constr_1`, and Paley construction II `Paley_constr_2`, of Hadamard matrices. Paley's constructions use so-called "Jacobsthal matrices", which will be introduced first in the first section.

- Throughout this chapter, `H` denotes a square matrix and in most parts has type `matrix I I ℚ`. In the context outside the codes, $n \in \mathbb{Q}$ denotes the cardinality of `I`.
- F denotes a finite field, and p denotes the characteristic of F, and q denotes the cardinality of F. F may also be denoted as $GF(q)$. J denotes the Jacobsthal matrix for F.
- The implementation in this chapter mainly follows [23] and also uses [11] as a reference.
- Please see file MAIN2.LEAN for the complete code work of this chapter's topic.

7.1 Jacobsthal matrix

The environment setting for this section:

```
variables {F : Type*} [field F] [fintype F] [decidable_eq F] {p : ℕ} [char_p F p]
local notation `q` := fintype.card F
```

DEFINITION 7.1.1. *The **Jacobsthal matrix** J for $GF(q)$ is the $q \times q$ matrix with rows and columns indexed by the elements of $GF(q)$ so that the entry in row a and column b is $\chi(a-b)$.*

```
variable (F) -- makes `F` an explicit variable to `Jacobsthal_matrix`.

@[reducible] def Jacobsthal_matrix : matrix F F ℚ := λ a b, χ (a-b)
-- We will use `J` to denote `Jacobsthal_matrix F` in annotations.
```

All the other implementations in this section are under the namespace `Jacobsthal_matrix`.

We build basic APIs and lemmas for Jacobsthal matrices:

```
/-- `J` is the circulant matrix `cir χ`. -/
lemma eq_cir : (Jacobsthal_matrix F) = cir χ := rfl

variable {F} -- this line makes `F` an implicit variable to the following lemmas/defs

@[simp] lemma diag_entry_eq_zero (i : F) :
(Jacobsthal_matrix F) i i = 0

@[simp] lemma non_diag_entry_eq {i j : F} (h : i ≠ j):
(Jacobsthal_matrix F) i j = 1 ∨ (Jacobsthal_matrix F) i j = -1

@[simp] lemma non_diag_entry_square_eq {i j : F} (h : i ≠ j):
(Jacobsthal_matrix F) i j * (Jacobsthal_matrix F) i j = 1

@[simp] lemma entry_square_eq (i j : F) :
(Jacobsthal_matrix F) i j * (Jacobsthal_matrix F) i j = ite (i=j) 0 1
```

LEMMA 7.1.2. *When $p \neq 2$, we have $J \cdot J^T = qI - \mathbb{1}$.*

PROOF. For $i, j \in F$,

$$(J \cdot J^T)\ i\ j = \sum_{k \in F} \chi(i-k)\chi(j-k) = \begin{cases} \sum\limits_{k \in F} \chi^2(i-k) = q-1 & \text{if } i = j \\ -1 & \text{if } i \neq j, \text{ by Lemma 4.3.7} \end{cases}$$

$$(qI - 1)\ i\ j = \begin{cases} q - 1 & \text{if } i = j \\ -1 & \text{otherwise} \end{cases}$$

Thus $J \cdot J^T = qI - \mathbb{1}$.

```
-- JJᵀ = qI - all_one
lemma mul_transpose_self (hp : p ≠ 2) :
(Jacobsthal_matrix F) ⬝ (Jacobsthal_matrix F)ᵀ = (q : ℚ) • 1 - all_one :=
begin
  ext i j,
  simp [mul_apply, all_one, Jacobsthal_matrix, one_apply],
  -- the current goal is
  -- ∑ (x : F), χ (i - x) * χ (j - x) = ite (i = j) q 0 - 1
  by_cases i = j,
  -- when i = j
  { simp[h, sum_ite, filter_ne, fintype.card],
    rw [@card_erase_of_mem' _ _ j (@finset.univ F _) _];
    simp },
  -- when i ≠ j
  simp [quad_char_sum_mul h hp, h],
end
```

■

Lemma 7.1.3. *When $p \neq 2$, we have $J \cdot \mathbb{1} = 0$.*

Proof. For $i, j \in F$,

$$\begin{aligned}(J \cdot \mathbb{1})\ i\ j &= \sum_{k \in F} \chi(i - k) \\ &= 0 \text{ by Lemma 4.3.6.}\end{aligned}$$

```
-- J ⬝ all_one = 0
@[simp] lemma mul_all_one (hp : p ≠ 2) :
(Jacobsthal_matrix F) ⬝ (all_one : matrix F F ℚ) = 0 :=
begin
  ext i j,
  simp [all_one, Jacobsthal_matrix, mul_apply],
  -- the current goal: ∑ (x : F), χ (i - x) = 0
  exact quad_char.sum_in_univ_eq_zero_reindex_1 hp,
end
```

■

Similarly, we can establish

Lemma 7.1.4. *When $p \neq 2$, $\mathbb{1} \cdot J = 0$.*

```
-- all_one ⬝ J = 0
@[simp] lemma all_one_mul (hp : p ≠ 2) :
(all_one : matrix F F ℚ) ⬝ (Jacobsthal_matrix F) = 0 :=
begin
  ext i j,
  simp [all_one, Jacobsthal_matrix, mul_apply],
  -- the current goal: ∑ (x : F), χ (x - j) = 0
  exact quad_char.sum_in_univ_eq_zero_reindex_2 hp,
end
```

and further two APIs:

```
-- J · col 1 = 0
@[simp] lemma mul_col_one (hp : p ≠ 2) :
Jacobsthal_matrix F · col 1 = 0

-- row 1 · Jᵀ = 0
@[simp] lemma row_one_mul_transpose (hp : p ≠ 2) :
row 1 · (Jacobsthal_matrix F)ᵀ = 0
```

In "most" cases, the Jacobsthal matrix J turns out to be either symmetric or skew-symmetric.

LEMMA 7.1.5. *If $q \equiv 1 \pmod 4$, then J is a symmetric matrix.*
PROOF.

$$\begin{aligned} J \text{ is symmetric} &\Leftrightarrow \forall i, j \in F \,.\, J\ j\ i = J\ i\ j \\ &\Leftrightarrow \forall i, j \in F \,.\, \chi(j-i) = \chi(i-j) \quad \text{which is the statement of Lemma 4.3.5} \end{aligned}$$

```
lemma is_sym_of (h : q ≡ 1 [MOD 4]) :
(Jacobsthal_matrix F).is_sym :=
by ext; simp [Jacobsthal_matrix, quad_char_is_sym_of' h i j]
```

■

LEMMA 7.1.6. *If $q \equiv 3 \pmod 4$, then q is a skew-symmetric matrix.*
PROOF.

$$\begin{aligned} J \text{ is skew symmetric} &\Leftrightarrow \forall i, j \in F \,.\, J\ j\ i = -J\ i\ j \\ &\Leftrightarrow \forall i, j \in F \,.\, \chi(j-i) = -\chi(i-j) \quad \text{which is the statement of Lemma 4.3.5} \end{aligned}$$

```
lemma is_skewsym_of (h : q ≡ 3 [MOD 4]) :
(Jacobsthal_matrix F).is_skewsym :=
by ext; simp [Jacobsthal_matrix, quad_char_is_skewsym_of' h i j]
```

■

We implement an API of `is_skewsym_of` for rewriting.

```
lemma is_skesym_of' (h : q ≡ 3 [MOD 4]) :
(Jacobsthal_matrix F)ᵀ = - (Jacobsthal_matrix F)
```

7.2 Paley construction I

DEFINITION 7.2.1. ***Paley construction I*** *is the block matrix $\begin{bmatrix} 1 & -\vec{1}^T \\ \vec{1} & I+J \end{bmatrix}$, where J is the Jacobsthal matrix for a given finite field F.*

```
variable (F)
def Paley_constr_1 : matrix (unit ⊕ F) (unit ⊕ F) ℚ :=
(1 : matrix unit unit ℚ).from_blocks (- row 1) (col 1) (1 + (Jacobsthal_matrix F))
```

THEOREM 7.2.2. *If $q \equiv 3 \pmod 4$, Paley construction I is a Hadamard matrix.*
PROOF. Suppose $q \equiv 3 \pmod 4$. Let $H = \begin{bmatrix} 1 & -\vec{1}^T \\ \vec{1} & I+J \end{bmatrix}$ be Paley construction I.

- As the diagonal of the Jacobsthal matrix J is $\vec{0}$, H obviously has entries equal to 1 or -1.
- It suffices to prove $H \cdot H^T$ is diagonal ($*$ is an omitted entry).

$$\begin{aligned} H \cdot H^T &= \begin{bmatrix} 1 & -\vec{1}^T \\ \vec{1} & I+J \end{bmatrix} \begin{bmatrix} 1 & \vec{1}^T \\ -\vec{1} & I+J^T \end{bmatrix} \\ &= \begin{bmatrix} * & \vec{1}^T - \vec{1}^T - \vec{1}^T \cdot J^T \\ \vec{1} - \vec{1} - J \cdot \vec{1} & \mathbb{1} + I + J + J^T + JJ^T \end{bmatrix} \end{aligned}$$

$$= \begin{bmatrix} * & 0-0 \\ 0-0 & \mathbb{1}+I+J+J^T+qI-\mathbb{1} \end{bmatrix} \quad \text{as } \vec{1}^T \cdot J^T = 0 \text{ and } J \cdot \vec{1} = 0 \text{ and } J \cdot J^T = qI - \mathbb{1}$$
$$= \begin{bmatrix} * & 0 \\ 0 & (1+q)I + J + J^T \end{bmatrix}$$
$$= \begin{bmatrix} * & 0 \\ 0 & (1+q)I \end{bmatrix} \quad \text{as } J \text{ is skew-symmetric when } q \equiv 3 \pmod 4$$

Thus, $H \cdot H^T$ is diagonal.

```
@[instance]
theorem Hadamard_matrix.Paley_constr_1 (h : q ≡ 3 [MOD 4]):
Hadamard_matrix (Paley_constr_1 F) :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- derive the char p of F
  resetI, -- resets the instance cache
  obtain ⟨hp, h'⟩ := char_ne_two' p h, -- prove p ≠ 2
  refine {..},
  -- first goal
  {
    rintros (i | i)  (j | j),
    all_goals {simp [Paley_constr_1, one_apply, Jacobsthal_matrix]},
    {by_cases i = j; simp*}
  },
  -- second goal
  rw ←mul_tranpose_is_diagonal_iff_has_orthogonal_rows,
  -- the above line changes the goal to prove J · Jᵀ is diagonal
  simp [Paley_constr_1, from_blocks_transpose, from_blocks_multiply,
        matrix.add_mul, matrix.mul_add, col_one_mul_row_one],
  rw [mul_col_one hp, row_one_mul_transpose hp, mul_transpose_self hp],
  simp,
  convert is_diagnoal_of_block_conditions ⟨is_diagonal_of_unit _, _, rfl, rfl⟩,
  -- to show the lower right corner block is diagonal
  {rw [is_skesym_of' h, add_assoc, add_comm, add_assoc], simp},
  any_goals {assumption},
end
```

■

Furthermore, Paley construction I is a special type Hadamard matrix:

THEOREM 7.2.3. *If $q \equiv 3 \pmod 4$, Paley construction I is a skew Hadamard matrix.*

PROOF. Again, suppose $q \equiv 3 \pmod 4$ and let $H = \begin{bmatrix} 1 & -\vec{1}^T \\ \vec{1} & I+J \end{bmatrix}$. Then,

$$H^T + H = \begin{bmatrix} 1 & \vec{1}^T \\ -\vec{1}^T & I + J^T \end{bmatrix} + \begin{bmatrix} 1 & -\vec{1}^T \\ \vec{1}^T & I + J \end{bmatrix}$$
$$= \begin{bmatrix} 2 & 0 \\ 0 & 2I + J^T + J \end{bmatrix}$$
$$= \begin{bmatrix} 2 & 0 \\ 0 & 2I \end{bmatrix} \quad \text{as } J \text{ is skew-symmetric}$$
$$= 2I$$

```
theorem Hadamard_matrix.Paley_constr_1_is_skew (h : q ≡ 3 [MOD 4]):
@is_skew _ _ (Paley_constr_1 F) (Hadamard_matrix.Paley_constr_1 h) _ :=
begin
  simp [is_skew, Paley_constr_1, from_blocks_transpose,
        from_blocks_add, is_skesym_of' h],
  have : 1 + -Jacobsthal_matrix F + (1 + Jacobsthal_matrix F) = 1 + 1,
  {noncomm_ring},
  rw [this], clear this,
  ext (a | i) (b | j),
  swap 3, rintro (b | j),
  any_goals {simp [one_apply, from_blocks, bit0]},
end
```

■

7.3 Paley construction II

Definition 7.3.1. ***Paley construction II*** *is the block matrix obtained by replacing all 0 entries in* $\begin{bmatrix} 0 & -\vec{1}^T \\ -\vec{1} & J \end{bmatrix}$ *with the matrix* $\begin{bmatrix} 1 & -1 \\ -1 & -1 \end{bmatrix}$*, and and all entries* ± 1 *with the matrix* $\pm\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$.

Before implementing this definition, We first define `C` and `D` as abbreviations of $\begin{bmatrix} 1 & -1 \\ -1 & -1 \end{bmatrix}$ and $\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$ respectively, and `E` as an abbreviation of $\begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}$, which equals the square of `C` and also the square of `D` (to avoid naming conflicts, all the helpers for implementing "Paley construction II" are under `namespace Paley_constr_2`):

```
variable (F)

def C : matrix (unit ⊕ unit) (unit ⊕ unit) ℚ :=
(1 : matrix unit unit ℚ).from_blocks (-1) (-1) (-1)

def D : matrix (unit ⊕ unit) (unit ⊕ unit) ℚ :=
(1 : matrix unit unit ℚ).from_blocks 1 1 (-1)

def E : matrix (unit ⊕ unit) (unit ⊕ unit) ℚ :=
(2 : matrix unit unit ℚ).from_blocks 0 0 2
```

and build some APIs for them, which turns out to be useful in the coming proofs:

```
variable (F)
/-- C is symmetric. -/
@[simp] lemma C_is_sym : C.is_sym

/-- D is symmetric. -/
@[simp] lemma D_is_sym : D.is_sym

/-- C · D = - D · C -/
lemma C_mul_D_anticomm : C · D = - D · C

/-- E is diagonal. -/
@[simp] lemma E_is_diagonal : E.is_diagonal

/-- C · C = E -/
@[simp] lemma C_mul_self : C · C = E

/-- C · Cᵀ = E -/
```

```
@[simp] lemma C_mul_transpose_self

/-- D · D = E -/
@[simp] lemma D_mul_self : D · D = E

/-- D · Dᵀ = E -/
@[simp] lemma D_mul_transpose_self : D · Dᵀ = E
```

To implement Paley construction II, we obviously need a notion of "replace". We implement "replace" as a one-ary operation that takes a matrix `A : matrix I J ℚ` and returns a matrix of type `matrix (I × (unit ⊕ unit)) (J × (unit ⊕ unit)) ℚ` consistent with the replacement in Definition 7.3.1:

```
variable (F)
def replace (A : matrix I J ℚ) :
matrix (I × (unit ⊕ unit)) (J × (unit ⊕ unit)) ℚ :=
λ ⟨i, a⟩ ⟨j, b⟩,
if (A i j = 0)
then C a b
else (A i j) • D a b
```

We want to establish some basic results about `replace`:

```
variable (F)
/-- `replace A` is a symmetric matrix if `A` is. -/
lemma replace_is_sym_of {A : matrix I I ℚ} (h : A.is_sym) :
(replace A).is_sym

/-- `replace 0 = I ⊗ C` -/
lemma replace_zero :
replace (0 : matrix unit unit ℚ) = 1 ⊗ C

/-- `replace A = A ⊗ D` for a matrix `A` with no `0` entries. -/
lemma replace_matrix_of_no_zero_entry
{A : matrix I J ℚ} (h : ∀ i j, A i j ≠ 0) : replace A = A ⊗ D

/-- In particular, we can apply `replace_matrix_of_no_zero_entry` to `- row 1`. -/
lemma replace_neg_row_one :
replace (-row 1 : matrix unit F ℚ) = (-row 1) ⊗ D

/-- `(replace 0) · (replace 0)ᵀ= I ⊗ E` -/
@[simp] lemma replace_zero_mul_transpose_self :
replace (0 : matrix unit unit ℚ) · (replace (0 : matrix unit unit ℚ))ᵀ = 1 ⊗ E

/-- `(replace A) · (replace A)ᵀ = (A · Aᵀ) ⊗ E` -/
@[simp] lemma replace_matrix_of_no_zero_entry_mul_transpose_self
{A : matrix I J ℚ} (h : ∀ i j, A i j ≠ 0) :
(replace A) · (replace A)ᵀ = (A · Aᵀ) ⊗ E
```

We illustrate two more non-obvious results about `replace`. We may write `replace` as r in the proofs in words:

Lemma 7.3.2. *replace*$(J) = J \otimes D + I \otimes C$.

Proof. For $i, j \in F$

- If $i = j$, then the (i, j)-block of *LHS* is C, and the (i, j)-block of *RHS* is $0 \bullet D + 1 \bullet C = C$.

- If $i \neq j$, then the (i,j)-block of LHS is $J(i,j) \bullet D$, and the (i,j)-block of RHS is $J(i,j) \bullet D + 0 \bullet C = J(i,j) \bullet D$.

Thus, $LHS = RHS$.

```
variable (F)
/-- `replace J = J ⊗ D + I ⊗ C` -/
lemma replace_Jacobsthal :
replace (Jacobsthal_matrix F) =
(Jacobsthal_matrix F) ⊗ D + 1 ⊗ C:=
begin
  ext ⟨i, a⟩ ⟨j, b⟩,
  by_cases i = j, --inspect the diagonal and non-diagonal entries respectively
  any_goals {simp [h, Jacobsthal_matrix, replace, Kronecker]},
end
```

■

Lemma 7.3.3. *If $q \equiv 1 \pmod 4$, $(replace\ J) \cdot (replace\ J)^T = ((q+1)I - \mathbb{1}) \otimes E$.*

Proof.

$$
\begin{aligned}
& r(J) \cdot r^T(J) \\
&= r(J) \cdot r(J^T) \\
&= r(J) \cdot r(J) \ \text{ as } J \text{ is symmetric when } q = 1 \pmod 4 \\
&= (J \otimes D + I \otimes C)^2 \ \text{ by Lemma 7.3.2} \\
&= J^2 \otimes D^2 + J \otimes DC + J \otimes CD + I \otimes C^2 \\
&= J^2 \otimes E - J \otimes CD + J \otimes CD + I \otimes E \\
&= (J^2 + I) \otimes E \\
&= (J \cdot J^T + I) \otimes E \\
&= (qI - \mathbb{1} + I) \otimes E \ \text{ by Lemma 7.1.2} \\
&= \big((q+I)I - \mathbb{1}\big) \otimes E
\end{aligned}
$$

```
variable {F}
lemma replace_Jacobsthal_mul_transpose_self' (h : q ≡ 1 [MOD 4]) :
replace (Jacobsthal_matrix F) ⬝ (replace (Jacobsthal_matrix F))ᵀ =
((Jacobsthal_matrix F) ⬝ (Jacobsthal_matrix F)ᵀ + 1) ⊗ E :=
begin
  simp [transpose_replace, (is_sym_of h).eq],
  simp [replace_Jacobsthal, matrix.add_mul, matrix.mul_add,
        K_mul, C_mul_D_anticomm, add_K],
  noncomm_ring
end

/-- enclose `replace_Jacobsthal_mul_transpose_self'` by replacing `J ⬝ Jᵀ` with
    `qI - all_one` -/
@[simp]lemma replace_Jacobsthal_mul_transpose_self (h : q ≡ 1 [MOD 4]) :
replace (Jacobsthal_matrix F) ⬝ (replace (Jacobsthal_matrix F))ᵀ =
(((q : ℚ) + 1) • (1 : matrix F F ℚ) - all_one) ⊗ E :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- derives a character p of F
```

```
  resetI, -- resets the instance cache
  obtain hp := char_ne_two p (or.inl h),  -- hp: p ≠ 2
  simp [replace_Jacobsthal_mul_transpose_self' h, add_smul],
  rw [mul_transpose_self hp],
  congr' 1, noncomm_ring,
  assumption
end
```

■

Now, we have done enough preparation for implementing Paley construction II:

```
variable (F)
def Paley_constr_2 :=
(replace (0 : matrix unit unit ℚ)).from_blocks
(replace (- row 1))
(replace (- col 1))
(replace (Jacobsthal_matrix F))
```

LEMMA 7.3.4. *If $q \equiv 1 \pmod 4$, Paley construction II is a symmetric matrix.*

PROOF. As if $q \equiv 1 \pmod 4$, J is a symmetric matrix (r replaces a symmetric matrix with a symmetric matrix), and $\left(r(-\vec{1})\right)^T = r\left((-\vec{1})^T\right) = r\left(-\vec{1}^T\right)$, Paley construction II is a symmetric matrix.

```
variable {F}
/-- `Paley_constr_2 F` is a symmetric matrix when `card F ≡ 1 [MOD 4]`. -/
@[simp] lemma Paley_constr_2_is_sym (h : q ≡ 1 [MOD 4]) :
(Paley_constr_2 F).is_sym :=
begin
  convert is_sym_of_block_conditions ⟨_, _, _⟩,
  { simp [replace_zero] }, -- `0` is symmetric
  { apply replace_is_sym_of (is_sym_of h) }, -- `J` is symmetric
  { simp [transpose_replace] } -- `(replace (-row 1))ᵀ = replace (-col 1)`
end
```

■

We split the proof that Paley construction II has entries in $\{1, -1\}$ out of the main proof that it is a Hadamard matrix when $q \equiv 1 \pmod 4$ to reduce the length of the main proof. The implementation is nothing more than asking Lean to inspect each block of the construction.

```
variable (F)
/-- Every entry of `Paley_constr_2 F` equals `1` or `-1`. -/
lemma Paley_constr_2.one_or_neg_one :
∀ (i j : unit × (unit ⊕ unit) ⊕ F × (unit ⊕ unit)),
Paley_constr_2 F i j = 1 ∨ Paley_constr_2 F i j = -1 :=
begin
  rintros (⟨a, (u₁|u₂)⟩ | ⟨i, (u₁ | u₂)⟩) (⟨b, (u₃|u₄)⟩ | ⟨j, (u₃ | u₄)⟩),
  all_goals {simp [Paley_constr_2, one_apply, Jacobsthal_matrix, replace, C, D]},
  all_goals {by_cases i = j},
  any_goals {simp [h]},
end
```

Now, it comes to the "big" theorem. After having prepared enough basic definitions and APIs, the implementation of the main proof is straightforward.

THEOREM 7.3.5. *If $q \equiv 1 \pmod 4$, Paley construction II is a Hadamard matrix.*

PROOF. Let $P = \begin{bmatrix} r(0) & r(-\vec{1}^T) \\ r(-\vec{1}) & r(J) \end{bmatrix}$. We are to show P is a Hadamard matrix.

(1) Every entry of P is in $\{1,-1\}$, as every entry of $\begin{bmatrix} 0 & -\vec{1}^T \\ -\vec{1} & J \end{bmatrix}$ is replaced with a 2×2 matrix whose entries equal 1 or -1.

(2) It suffices to show $P\cdot P^T$ is diagonal ($*$ below is a block omitted, as $P\cdot P^T$ is symmetric).

$$
\begin{aligned}
P\cdot P^T &= \begin{bmatrix} 1\otimes E + q\otimes E & r(0)\cdot r^T(-\vec{1}) + r(-\vec{1}^T)\cdot r^T(J) \\ * & \mathbb{1}\otimes E + r(J)\cdot r^T(J) \end{bmatrix} \\
&= \begin{bmatrix} (1+q)\otimes E & r(0)\cdot r(-\vec{1}^T) + r(-\vec{1}^T)\cdot r(J^T) \\ * & \mathbb{1}\otimes E + r(J)\cdot r^T(J) \end{bmatrix} \\
&= \begin{bmatrix} (1+q)\otimes E & r(0)\cdot r(-\vec{1}^T) + r(-\vec{1}^T)\cdot r(J) \\ * & \mathbb{1}\otimes E + r(J)\cdot r^T(J) \end{bmatrix} \\
&\quad \text{as } J \text{ is symmetric if } q\equiv 1 \pmod 4 \\
&= \begin{bmatrix} (1+q)\otimes E & -(1\otimes C)(\vec{1}^T\otimes D) - (\vec{1}^T\otimes D)(J\otimes D + I\otimes C) \\ * & \mathbb{1}\otimes E + \big((q+1)I - \mathbb{1}\big)\otimes E \end{bmatrix} \\
&\quad \text{by Lemmas 7.3.2 and 7.3.3} \\
&= \begin{bmatrix} (1+q)\otimes E & -\vec{1}^T\otimes CD - \vec{1}^T J\otimes D^2 - \vec{1}^T\otimes DC \\ * & (q+1)I\otimes E \end{bmatrix} \\
&= \begin{bmatrix} (1+q)\otimes E & -\vec{1}^T\otimes CD - \vec{0}^T\otimes E + \vec{1}^T\otimes CD \\ * & (q+1)I\otimes E \end{bmatrix} \\
&= \begin{bmatrix} (1+q)\otimes E & 0 \\ * & (q+1)I\otimes E \end{bmatrix}
\end{aligned}
$$

As E and I are diagonal, so are $(1+q)\otimes E$ and $(q+1)I\otimes E$. Then, as $P\cdot P^T$ is symmetric, it is diagonal.

```
variable {F}
@[instance]
theorem Hadamard_matrix.Paley_constr_2 (h : q ≡ 1 [MOD 4]):
Hadamard_matrix (Paley_constr_2 F) :=
begin
  refine {..},
  -- the first goal
  { exact Paley_constr_2.one_or_neg_one F },
  -- the second goal
  -- turns the goal to `Paley_constr_2 F · (Paley_constr_2 F)ᵀ` is diagonal
  rw ←mul_tranpose_is_diagonal_iff_has_orthogonal_rows,
  -- sym : `Paley_constr_2 F · (Paley_constr_2 F)ᵀ` is symmetric
  have sym := mul_transpose_self_is_sym (Paley_constr_2 F),
  -- The next `simp` turns `Paley_constr_2 F · (Paley_constr_2 F)ᵀ` into a block form.
  simp [Paley_constr_2, from_blocks_transpose, from_blocks_multiply] at *,
  convert is_diagnoal_of_sym_block_conditions sym ⟨_, _, _⟩, -- splits into the three goals
  any_goals {clear sym},
  -- to prove the upper left corner block is diagonal.
  { simp [row_one_mul_col_one, ← add_K],
    apply K_is_diagonal_of; simp },
  -- to prove the lower right corner block is diagonal.
  { simp [h, col_one_mul_row_one, ← add_K],
    apply smul_is_diagonal_of,
    apply K_is_diagonal_of; simp },
  -- to prove the upper right corner block is `0`.
  { obtain ⟨p, inst⟩ := char_p.exists F, -- obtains the character p of F
```

```
    resetI, -- resets the instance cache
    obtain hp := char_ne_two_of p (or.inl h), -- hp: p ≠ 2
    simp [transpose_replace, (is_sym_of h).eq],
    simp [replace_zero, replace_neg_row_one, replace_Jacobsthal,
          matrix.mul_add, K_mul, C_mul_D_anticomm],
    rw [←(is_sym_of h).eq, row_one_mul_transpose hp],
    simp, assumption }
end
```

■

8 A HADAMARD MATRIX OF ORDER 92

In Chapter 10, I will illustrate why, at this point, we have constructed at least one Hadamard matrix of order n for all possible natural numbers $n \leq 112$ except $n = 92$. In this chapter, we construct a Hadamard matrix of order 92. The construction of this matrix is originally given in [3], and we prove this matrix is a Hadamard matrix using the Williamson's method given in [24]. This construction uses (symmetric) circulant matrices, of which mathlib has not encoded the notion. As such, we are to introduce the implementation of circulant matrices in the first section. Please see file MAIN2.LEAN for the complete code work of the implementation of this order 92 matrix.

8.1 Circulant matrix

DEFINITION 8.1.1. *A $n \times n$ **circulant matrix** C is a matrix of the form*

$$C = \begin{bmatrix} c_0 & c_{n-1} & \cdots & c_2 & c_1 \\ c_1 & c_0 & c_{n-1} & & c_2 \\ \vdots & c_1 & c_0 & \ddots & \vdots \\ c_{n-2} & & \ddots & \ddots & c_{n-1} \\ c_{n-1} & c_{n-2} & \cdots & c_1 & c_0 \end{bmatrix}$$

Thus, every $n \times n$ circulant matrix C is generated by a vector v of size n. Also, one needs a notion of "subtraction" over the set/type, which the entries of v are in. We implement circulant matrices as follows:

```
/-- Given the condition `[has_sub I]` and a vector `v : I → α`,
    we define `cir v` to be the circulant matrix generated by `v` of type `matrix I I α`. -/
def cir [has_sub I] (v : I → α) : matrix I I α
| i j := v (i - j)
```

Here are some selected results on circulant matrices (please see CIRCULANT_MATRIX.LEAN for the complete implementation work of circulant matrices):

```
/-- When `I` is an `add_group`, the 0th column of `cir v` is `v`. -/
lemma cir_col_zero_eq [add_group I] (v : I → α) :
(λ i, (cir v) i 0) = v := by ext; simp [cir]

/-- When `I` is an `add_group`, `cir v = cir w ↔ v = w`. -/
lemma cir_ext_iff [add_group I] {v w : I → α} :
cir v = cir w ↔ v = w

lemma fin.cir_ext_iff {v w : fin n → α} :
cir v = cir w ↔ v = w

/-- The sum of two circulant matrices `cir v` and `cir w`
    is also a circulant matrix `cir (v + w)`. -/
lemma cir_add [has_add α] [has_sub I] (v w : I → α) :
cir v + cir w = cir (v + w)

/-- The product of two circulant matrices `cir v` and `cir w`
    is also a circulant matrix `cir (mul_vec (cir w) v)`. -/
lemma cir_mul [comm_semiring α] [add_comm_group I] (v w : I → α) :
cir v · cir w = cir (mul_vec (cir w) v)

lemma fin.cir_mul [comm_semiring α] (v w : fin n → α) :
cir v · cir w = cir (mul_vec (cir w) v)
```

```
/-- `k • cir v` is another circluant matrix `cir (k • v)`. -/
lemma smul_cir [has_sub I] [has_scalar R α] {k : R} {v : I → α} :
k • cir v = cir (k • v)

/-- The identity matrix is a circulant matrix. -/
lemma one_eq_cir [has_zero α] [has_one α] [decidable_eq I] [add_group I]:
(1 : matrix I I α) = cir (λ i, ite (i = 0) 1 0)

/-- Given a set `S`, every entry of `cir v` is in `S` if every entry of `v` is in `S`. -/
lemma cir_entry_in_of_vec_entry_in [has_sub I] {S : set α} {v : I → α} :
(∀ k, v k ∈ S) → ∀ i j, (cir v) i j ∈ S

/-- The circulant matrix `cir v` is symmetric iff `∀ i j, v (j - i) = v (i - j)`. -/
lemma cir_is_sym_ext_iff' [has_sub I] {v : I → α} :
(cir v).is_sym ↔ ∀ i j, v (j - i) = v (i - j)

/-- The circulant matrix `cir v` is symmetric iff `v (- i) = v i` if `[add_group I]`. -/
lemma cir_is_sym_ext_iff [add_group I] {v : I → α} :
(cir v).is_sym ↔ ∀ i, v (- i) = v i

lemma fin.cir_is_sym_ext_iff {v : fin n → α} :
(cir v).is_sym ↔ ∀ i, v (- i) = v i
```

- Some of the implemented results (in fact, most of) require a stronger condition for `I` than just `[has_sub I]`.
- `fin.foo` is the `fin` version of `foo`. Namely, the index type of the circulant matrices in discussion is `fin n`, which is the subtype of $\mathbb{N}$ consisting of natural numbers strictly smaller than `n` : $\mathbb{N}$.
- One of these APIs worth being pointed out is `cir_is_sym_ext_iff`, which proves `cir v` is symmetric if and only if `v` is "symmetric" (i.e. $\forall i, v(-i) = v(i)$) when `I` is an additive group, because the circulant matrices that will be used for constructing the order 92 matrix are all symmetric matrices.

8.2 The matrix

This section gives the construction of the order 92 matrix, which we will denote by H (and by `H_92` in Lean), and a proof that this gives an Hadamard matrix.

We first have to encode the data used in the construction of H.

Definition 8.2.1. *We define A, B, C, D to be the circulant matrices generated by vectors a, b, c, d of size 23 given as follows:*

```
def a : fin 23 → ℚ :=
![ 1,  1, -1, -1, -1,  1, -1, -1, -1,  1, -1,  1,  1, -1,  1, -1, -1, -1,  1, -1, -1, -1,  1]
def b : fin 23 → ℚ :=
![ 1, -1,  1,  1, -1,  1,  1, -1, -1,  1,  1,  1,  1,  1,  1, -1, -1,  1,  1, -1,  1,  1, -1]
def c : fin 23 → ℚ :=
![ 1,  1,  1, -1, -1, -1,  1,  1, -1,  1, -1,  1,  1, -1,  1, -1,  1,  1, -1, -1, -1,  1,  1]
def d : fin 23 → ℚ :=
![ 1,  1,  1, -1,  1,  1,  1, -1,  1, -1, -1, -1, -1, -1, -1,  1, -1,  1,  1,  1, -1,  1,  1]

abbreviation A := cir a
abbreviation B := cir b
abbreviation C := cir c
abbreviation D := cir d
```

In mathlib, the notation `![v₁, v₂, ...]` is used to denote vectors with index type `fin n` for some natural number n.

We note that a, b, c, d are "symmetric", and have entries as 1 or -1; and thus A, B, C, D are symmetric, and have entries as 1 or -1.

More importantly, A, B, C, D are chosen specially to be a solution of the equation $A\cdot A+B\cdot B+C\cdot C+D\cdot D = 92I$. With the solution at hand, the proof of $A\cdot A+B\cdot B+C\cdot C+D\cdot D = 92I$ can be a direct computation. However, unlike some computer algebra systems, as an interactive proof assistant, Lean has relatively poor ability of doing computation (indeed, `#eval A · Aᵀ` has already taken an extremely large amount of time!). As such, we would like to use some tricks to reduce the amount of computation work that we require Lean to do. Because the product of two circulant matrices is circulant, and the sum of two circulant matrices is circulant, the LHS of the equation is a circulant matrix, generated by $A\cdot a+B\cdot b+C\cdot c+D\cdot d$; and obviously, the RHS of $92I$ is also a circulant matrix. As such, we only need to compute $A\cdot a+B\cdot b+C\cdot c+D\cdot d$ (LHS) and compare it with the first column of $92I$ (RHS); namely, to compare the 23 pairs of entries. The computation amount is hugely reduced. Even though, to compute the 23 entries of the LHS is too hard for Lean to do in a single proof. As such, I intentionally break the 23 computations into 23 auxiliary lemmas, each of which computes one entry of the LHS and compare it to the corresponding entry of the RHS. Every auxiliary lemma is named `eq_auxᵢ`, which computes and compares the i-th (indexed from 0) entries of LHS and RHS.

```
@[simp] lemma eq_aux₀:
dot_product (λ (j : fin 23), a (0 - j)) a +
dot_product (λ (j : fin 23), b (0 - j)) b +
dot_product (λ (j : fin 23), c (0 - j)) c +
dot_product (λ (j : fin 23), d (0 - j)) d = 92 :=
by {unfold a b c d, norm_num}

@[simp] lemma eq_aux₁:
dot_product (λ (j : fin 23), a (1 - j)) a +
dot_product (λ (j : fin 23), b (1 - j)) b +
dot_product (λ (j : fin 23), c (1 - j)) c +
dot_product (λ (j : fin 23), d (1 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₂:
dot_product (λ (j : fin 23), a (2 - j)) a +
dot_product (λ (j : fin 23), b (2 - j)) b +
dot_product (λ (j : fin 23), c (2 - j)) c +
dot_product (λ (j : fin 23), d (2 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

...... (omitted)

@[simp] lemma eq_aux₂₂:
dot_product (λ (j : fin 23), a (22 - j)) a +
dot_product (λ (j : fin 23), b (22 - j)) b +
dot_product (λ (j : fin 23), c (22 - j)) c +
dot_product (λ (j : fin 23), d (22 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}
```

where `fin_23_shift` is an API used for normalising vectors

```
/-- `fin_23_shift` normalizes `λ (j : fin 23), f (s j)` in `![]` form,
    where `s : fin 23 → fin 23` is a function shifting indices. -/
lemma fin_23_shift (f : fin 23 → ℚ) (s : fin 23 → fin 23) :
(λ (j : fin 23), f (s j)) =
![f (s 0), f (s 1), f (s 2), f (s 3), f (s 4), f (s 5), f (s 6), f (s 7),
  f (s 8), f (s 9), f (s 10), f (s 11), f (s 12), f (s 13), f (s 14), f (s 15),
  f (s 16), f (s 17), f (s 18), f (s 19), f (s 20), f (s 21), f (s 22)] :=
by {ext i, fin_cases i, any_goals {simp},}
```

We can now establish the equality

Lemma 8.2.2. $A \cdot A + B \cdot B + C \cdot C + D \cdot D = 92I$

```
lemma equality :
A · A + B · B + C · C + D · D = (92 : ℚ) • (1 : matrix (fin 23) (fin 23) ℚ) :=
begin
  -- the first `simp` transfers the equation to the form `cir .. = cir ..`
  simp [cir_mul, cir_add, one_eq_cir, smul_cir],
  -- we then show the two `cir`s consume equal arguments
  congr' 1,
  -- to show the two vectors are equal
  ext i,
  simp [mul_vec, cir],
  -- ask lean to inspect the 23 pairs entries one by one
  fin_cases i,
  exact eq_aux₀,
  exact eq_aux₁,
  ... (omitted)
  exact eq_aux₂₂,
end
```

To construct H, we need more encoded data.

Definition 8.2.3. *We define i, j, k to be the three skew-symmetric matrices:*

$$i = \begin{bmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \qquad j = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \end{bmatrix}$$

$$k = ij = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \end{bmatrix}$$

implemented as

```
def i : matrix (fin 4) (fin 4) ℚ :=
![![0, 1, 0, 0],
  ![-1, 0, 0, 0],
  ![0, 0, 0, -1],
  ![0, 0, 1, 0]]

def j : matrix (fin 4) (fin 4) ℚ :=
![![0, 0, 1, 0],
  ![0, 0, 0, 1],
```

```
  ![-1, 0, 0, 0],
  ![0, -1, 0, 0]]

def k: matrix (fin 4) (fin 4) ℚ :=
![![0, 0, 0, 1],
  ![0, 0, -1, 0],
  ![0, 1, 0, 0],
  ![-1, 0, 0, 0]]
```

The four matrices i, j, k, and 1(the identity matrix of order 4) are isomorphic to the units of quaternions and satisfy the usual equations.

Now, we are ready to reveal the mysterious H:

DEFINITION 8.2.4. *We define matrix $H = A \otimes 1 + B \otimes i + C \otimes j + D \otimes k$, where 1 denotes the identity matrix of order 4. We note H has order 92.*

```
def H_92 := A ⊗ 1 + B ⊗ i + C ⊗ j + D ⊗ k
```

THEOREM 8.2.5. *H is a Hadamard matrix.*

PROOF. 1 denotes the identity matrix of order 4 in the expression of H.

(1) As

$$H = A \otimes 1 + B \otimes i + C \otimes j + D \otimes k = \begin{bmatrix} A & B & C & D \\ -B & A & -D & C \\ -C & D & A & -B \\ -D & -C & B & A \end{bmatrix}$$

and every entry of A, B, C, D is in $\{1, -1\}$, every entry of H is in $\{1, -1\}$.

(2) It suffices to show $H \cdot H^T$ is diagonal.

$$\begin{aligned} H \cdot H^T &= (A \otimes 1 + B \otimes i + C \otimes j + D \otimes k) \cdot \\ &\quad (A^T \otimes 1^T + B^T \otimes i^T + C^T \otimes j^T + D^T \otimes k^T) \\ &= (A \otimes 1 + B \otimes i + C \otimes j + D \otimes k) \cdot \\ &\quad (A \otimes 1 - B \otimes i - C \otimes j - D \otimes k) \\ &\quad \text{as } A, B, C, D \text{ are symmetric, } i, j, k \text{ are skew-symmetric} \\ &= AA \otimes 1 - AB \otimes i - AC \otimes j - AD \otimes k + \\ &\quad AB \otimes i + BB \otimes 1 - BC \otimes k + BD \otimes j + \\ &\quad AC \otimes j + BC \otimes k + CC \otimes 1 - CD \otimes i + \\ &\quad AD \otimes k - BD \otimes j + CD \otimes i + DD \otimes 1 \\ &= A^2 \otimes 1 + B^2 \otimes 1 + C^2 \otimes 1 + D^2 \otimes 1 \\ &= (A^2 + B^2 + C^2 + D^2) \otimes 1 \\ &= 92I \otimes 1 \quad \text{by Lemma 8.2.2} \end{aligned}$$

which is diagonal.

```
/-- Proves every entry of `H_92` is `1` or `-1`. -/
lemma H_92.one_or_neg_one : ∀ i j, (H_92 i j) = 1 ∨ (H_92 i j) = -1 :=
begin
  rintros ⟨c, a⟩ ⟨d, b⟩,
  simp [H_92, Kronecker],
  fin_cases a,
  any_goals {fin_cases b},
```

```
  any_goals {norm_num [one_apply, i, j, k]},
end

/-- Proves `H_92 · H_92ᵀ` is a diagonal matrix. -/
lemma H_92_mul_transpose_self_is_diagonal : (H_92 · H_92ᵀ).is_diagonal :=
begin
  simp [H_92, transpose_K, matrix.mul_add, matrix.add_mul, K_mul,
  cir_mul_comm _ a, cir_mul_comm c b, cir_mul_comm d b, cir_mul_comm d c],
  have :
  (cir a · cir a)⊗1 + -(cir a · cir b)⊗i + -(cir a · cir c)⊗j + -(cir a · cir d)⊗k +
  ((cir a · cir b)⊗i + (cir b · cir b)⊗1 + -(cir b · cir c)⊗k + (cir b · cir d)⊗j) +
  ((cir a · cir c)⊗j + (cir b · cir c)⊗k + (cir c · cir c)⊗1 + -(cir c · cir d)⊗i) +
  ((cir a · cir d)⊗k + -(cir b · cir d)⊗j + (cir c · cir d)⊗i + (cir d · cir d)⊗1) =
  (cir a · cir a)⊗1 + (cir b · cir b)⊗1 + (cir c · cir c)⊗1 + (cir d · cir d)⊗1 :=
  by abel,
  rw this, clear this,
  simp [←add_K, equality], -- uses `equality`
end

@[instance] theorem Hadamard_matrix.H_92 : Hadamard_matrix H_92 :=
⟨H_92.one_or_neg_one,
mul_tranpose_is_diagonal_iff_has_orthogonal_rows.1 H_92_mul_transpose_self_is_diagonal⟩
```

■

9 ORDERS OF HADAMARD MATRICES

We have established two results Theorem 6.1.4 and Theorem 6.2.3 in chapter 6 about the possible orders of Hadamard matrices. In this chapter, we show that the order of a Hadamard matrix can only be 0, 1, 2, or a multiple of 4, and encode the statement of the Hadamard conjecture, which is the converse: a Hadamard matrix of order $4k$ exists for every positive integer k.

- Throughout this chapter, `H` denotes a square matrix and in most parts has type `matrix I I ℚ`. In the context outside the codes, $n \in \mathbb{Q}$ denotes the cardinality of `I`.
- The implementation in this chapter follows [20] and [11].
- Please see file MAIN2.LEAN for the complete code work of this chapter's topic.

9.1 An order constraint

THEOREM 9.1.1. *If a Hadamard matrix H of order n exists, and $n \geqslant 3$, then n must be a multiple of 4.*

PROOF. Suppose $n \geqslant 3$, then there are three distinct rows i_1, i_2, i_3 of H. Without loss of generality we may suppose H is normalised and i_1 is the first row for the convenience of presentation (in the Lean proof, we don't need this assumption). Then the composition of these three rows is as follows up to a permutation of columns (− denotes −1):

$$\underbrace{\begin{matrix}1 & 1 & \cdots & 1\\ 1 & 1 & \cdots & 1\\ 1 & 1 & \cdots & 1\end{matrix}}_{i} \quad \underbrace{\begin{matrix}1 & 1 & \cdots & 1\\ 1 & 1 & \cdots & 1\\ - & - & \cdots & -\end{matrix}}_{j} \quad \underbrace{\begin{matrix}1 & 1 & \cdots & 1\\ - & - & \cdots & -\\ 1 & 1 & \cdots & 1\end{matrix}}_{k} \quad \underbrace{\begin{matrix}1 & 1 & \cdots & 1\\ - & - & \cdots & -\\ - & - & \cdots & -\end{matrix}}_{\ell}$$

By Lemma 5.2.5, we have

$$\begin{aligned} i + j &= k + \ell \\ i + k &= j + \ell \\ i + \ell &= j + k \end{aligned}$$

which imply $i = j = k = \ell$, and thus $n = 4i$, which is a multiple of 4.

```
theorem Hadamard_matrix.order_constraint
[decidable_eq I] (H : matrix I I ℚ) [Hadamard_matrix H]
: card I ≥ 3 → 4 ∣ card I :=
begin
  intros h, -- h: card I ≥ 3
  -- pick three distinct rows i₁, i₂, i₃
  obtain ⟨i₁, i₂, i₃, ⟨h₁₂, h₁₃, h₂₃⟩⟩:= pick_elements h,
  -- the cardinalities of J₁, J₂, J₃, J₄ are denoted as i, j, k, l in the proof in words
  set J₁ := {j : I | H i₁ j = H i₂ j ∧ H i₂ j = H i₃ j},
  set J₂ := {j : I | H i₁ j = H i₂ j ∧ H i₂ j ≠ H i₃ j},
  set J₃ := {j : I | H i₁ j ≠ H i₂ j ∧ H i₁ j = H i₃ j},
  set J₄ := {j : I | H i₁ j ≠ H i₂ j ∧ H i₂ j = H i₃ j},
  -- dₘₙ proves Jₘ Jₙ are disjoint
  have d₁₂: disjoint J₁ J₂,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros, linarith},
  have d₁₃: disjoint J₁ J₃,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d, exact c a},
  have d₁₄: disjoint J₁ J₄,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d, exact c a},
  have d₂₃: disjoint J₂ J₃,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d, exact c a},
  have d₂₄: disjoint J₂ J₄,
```

```
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d, exact c a},
  have d₃₄: disjoint J₃ J₄,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d,
  have : H i₁ x = H i₂ x, {linarith}, exact c this},
  -- u₁₂ proves J₁ ∪ J₂ = matched H i₁ i₂
  have u₁₂: J₁.union J₂ = matched H i₁ i₂,
  {ext, simp [J₁, J₂, matched, set.union], tauto},
  -- u₁₃ proves J₁ ∪ J₃ = matched H i₁ i₃
  have u₁₃: J₁.union J₃ = matched H i₁ i₃,
  {ext, simp [J₁, J₃, matched, set.union], by_cases g : H i₁ x = H i₂ x; simp [g]},
  -- u₁₄ proves J₁ ∪ J₄ = matched H i₂ i₃
  have u₁₄: J₁.union J₄ = matched H i₂ i₃,
  {ext, simp [J₁, J₄, matched, set.union], tauto},
  -- u₂₃ proves J₂ ∪ J₃ = mismatched H i₂ i₃
  have u₂₃: J₂.union J₃ = mismatched H i₂ i₃,
  { ext, simp [J₂, J₃, mismatched, set.union],
    by_cases g₁ : H i₂ x = H i₃ x; simp [g₁],
    by_cases g₂ : H i₁ x = H i₂ x; simp [g₁, g₂],
    exact entry_eq_entry_of (ne.symm g₂) g₁ },
  -- u₂₄ proves J₂ ∪ J₄ = mismatched H i₂ i₄
  have u₂₄: J₂.union J₄ = mismatched H i₁ i₃,
  { ext, simp [J₂, J₄, mismatched, set.union],
    by_cases g₁ : H i₁ x = H i₂ x; simp [g₁],
    split, {rintros g₂ g₃, exact g₁ (g₃.trans g₂.symm)},
    intros g₂,
    exact entry_eq_entry_of g₁ g₂ },
 -- u₃₄ proves J₃ ∪ J₄ = mismatched H i₁ i₂
  have u₃₄: J₃.union J₄ = mismatched H i₁ i₂,
  { ext, simp [J₃, J₄, matched, set.union],
    split; try {tauto},
    intros g₁,
    by_cases g₂ : H i₁ x = H i₃ x,
    { left, exact ⟨g₁, g₂⟩ },
    { right, exact ⟨g₁, entry_eq_entry_of g₁ g₂⟩ } },
  -- eq₁: |H.matched i₁ i₂| = |H.mismatched i₁ i₂|
  have eq₁ := card_match_eq_card_mismatch H h₁₂,
  -- eq₂: |H.matched i₁ i₃| = |H.mismatched i₁ i₃|
  have eq₂ := card_match_eq_card_mismatch H h₁₃,
  -- eq₃: |H.matched i₂ i₃| = |H.mismatched i₂ i₃|
  have eq₃ := card_match_eq_card_mismatch H h₂₃,
  -- eq : |I| = |H.matched i₁ i₂| + |H.mismatched i₁ i₂|
  have eq := card_match_add_card_mismatch H i₁ i₂,
  -- rewrite eq to |I| = |J₁| + |J₂| + |J₃| + |J₄|, and
  -- rewrite eq₁ to |J₁| + |J₂| = |J₃| + |J₄|
  rw [set.card_disjoint_union' d₁₂ u₁₂, set.card_disjoint_union' d₃₄ u₃₄] at eq₁ eq,
  -- rewrite eq₂ to |J₁| + |J₃| = |J₂| + |J₄|
  rw [set.card_disjoint_union' d₁₃ u₁₃, set.card_disjoint_union' d₂₄ u₂₄] at eq₂,
  -- rewrite eq₃ to |J₁| + |J₄| = |J₂| + |J₄|
  rw [set.card_disjoint_union' d₁₄ u₁₄, set.card_disjoint_union' d₂₃ u₂₃] at eq₃,
  -- g₂₁, g₃₁, g₄₁ prove that |J₁| = |J₂| = |J₃| = |J₄|
  have g₂₁ : J₂.card = J₁.card, {linarith},
  have g₃₁ : J₃.card = J₁.card, {linarith},
```

```
  have g₄₁ : J₄.card = J₁.card, {linarith},
  -- rewrite eq to |I| = |J₁| + |J₁| + |J₁| + |J₁|
  rw [g₂₁, g₃₁, g₄₁, set.univ_card_eq_fintype_card] at eq,
  use J₁.card,
  simp [eq], noncomm_ring,
end
```

■

9.2 Hadamard conjecture

The most important open question in the theory of Hadamard matrices is that of existence. The Hadamard conjecture states that a Hadamard matrix of order $4k$ exists for every positive integer k. We formalise the conjecture as follows:

```
theorem Hadamard_matrix.Hadamard_conjecture:
∀ k : ℕ, ∃ (I : Type*) [fintype I],
by exactI ∃ (H : matrix I I ℚ) [Hadamard_matrix H],
card I = 4 * k :=
sorry -- Here, `sorry` means if you ask me to prove this conjecture,
      -- then I have to apologize.
```

`sorry` is a tactic that magically closes any goals.

10 CONCLUDING THOUGHTS

As promised, I elaborate here on the claim that we showed the existence of an Hadamard matrix of order n for all possible $n \leq 112$. We implemented the trivial cases, namely when $n = 0,\ 1,\ 2$, in Section 5.3. For $3 \leq n \leq 112$, the possible values that n can take, by Theorem 9.1.1, are the multiplies of 4: 4, 8, 12,..., 112. Theorem 6.1.4 proves we implemented an infinite series of Hadamard matrices of order 2^k. Hence, it covers the cases: 4, 8, 16, 32, 64. Theorem 7.2.3 (Paley construction I) covers (we test whether $n - 1$ is a prime power): 12, 20, 24, 28, 44, 48, 60, 68, 72, 80, 84, 104, 108. Theorem 7.3.5 (Paley construction II) further covers (we test whether $n/2 - 1$ is a prime power and is congruent to 1 mod 4): 36, 57, 76, 100. What left now are 40, 56, 88, 92, 96, 112. Theorem 6.2.3 covers $40(= 2 * 20)$, $56(= 2 * 28)$, $88(= 2 * 44)$, $96(= 2 * 48)$, $112(= 4 * 28)$, and Chapter 8 implemented a Hadamard matrix of order 92.

This project opens the door to many future directions for mathlib contributors (including myself):

- Every regular symmetric Hadamard matrices with constant diagonal induces a strongly regular graph [4], [5, Sect.8.1]. I am working on extending the strongly regular graph part of mathlib currently. This part of mathlib is still at a very "infantile" stage. Unfortunately, due to the length constraint of the thesis, I am unable to show my ongoing work on graph theory.
- As mentioned in the remark after Lemma 4.2.8, one can naturally generalise Lemma 4.2.8 and Lemma 4.3.2 (mathematically trivial) to git rid of the condition $p \neq 2$.
- One can attempt to implement more constructions of Hadamard matrices (for examples, other constructions described in [24], a skew Hadamard matrix of order 92 in [19]).
- One familiar with coding theory can implement error correction codes based on Hadamard matrices.
- Many others.....

REFERENCES

[1] Jeremy Avigad, Kevin Buzzard, Robert Y. Lewis, and Patrick Massot. *Mathematics in Lean*. The Lean Community, 2021. URL: https://leanprover-community.github.io/mathematics_in_lean/mathematics_in_lean.pdf.

[2] Jeremy Avigad, Leonardo de Moura, and Soonho Kong. *Theorem Proving in Lean*. The Lean Community, 2021. URL: https://leanprover.github.io/theorem_proving_in_lean/theorem_proving_in_lean.pdf.

[3] Leonard Baumert, S.W. Golomb, and Marshall Hall Jr. Discovery of an Hadamard matrix of order 92. *Bulletin of the American Mathematical Society*, 68(3):237–238, 1962.

[4] Andries E. Brouwer and Willem H. Haemers. *Spectra of graphs*. Universitext. Springer, New York, 2012. doi:10.1007/978-1-4614-1939-6.

[5] Andries E. Brouwer and H. Van Maldeghem. *Strongly Regular Graphs*. Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2022. URL: https://homepages.cwi.nl/~aeb/math/srg/rk3/srgw.pdf.

[6] Nathann Cohen and Dmitrii V. Pasechnik. Implementing Brouwer's database of strongly regular graphs. *Designs, Codes and Cryptography*, 84(1-2):223–235, August 2016. doi:10.1007/s10623-016-0264-x.

[7] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

[8] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25*, pages 378–388, Cham, 2015. Springer International Publishing.

[9] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997. doi:10.1.1.37.74.

[10] Alena Gusakov, Bhavik Mehta, and Kyle A. Miller. Formalizing Hall's Marriage Theorem in Lean, 2021. arXiv:2101.00127.

[11] F.J. MacWilliams and N.J.A. Sloane. Nonlinear codes, Hadamard matrices, designs and the Golay code. In *The Theory of Error-Correcting Codes*, volume 16 of *North-Holland Mathematical Library*, pages 38–79. Elsevier, 1977. doi:10.1016/S0924-6509(08)70527-0.

[12] The mathlib Community. The lean mathematical library. *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, Jan 2020. URL: http://dx.doi.org/10.1145/3372885.3373824, doi:10.1145/3372885.3373824.

[13] Michael Shulman. Homotopy type theory: the logic of space, 2017. e-print 1703.03007, arXiv.org.

[14] The Agda Development Team. *Agda programming language*, 2021. URL: https://wiki.portal.chalmers.se/agda/pmwiki.php.

[15] The Coq Development Team. *The Coq Proof Assistant*, January 2021. doi:10.5281/zenodo.4501022.

[16] The mathlib Community. Attributes — mathlib docs, 2021. URL: https://leanprover-community.github.io/mathlib_docs/attributes.html.

[17] The mathlib Community. Mathlib tactics — mathlib docs, 2021. URL: https://leanprover-community.github.io/mathlib_docs/tactics.html.

[18] The mathlib Community. Simplifier - mathlib docs, 2021. URL: https://leanprover-community.github.io/extras/simp.html.

[19] Jennifer Wallis. A skew-Hadamard matrix of order 92. *Bulletin of the Australian Mathematical Society*, 5(2):203–204, 1971. doi:10.1017/S0004972700047079.

[20] Wikipedia. Hadamard matrix, 2021. URL: https://en.wikipedia.org/w/index.php?title=Hadamard%20matrix.

[21] Wikipedia. Hadamard product (matrices), 2021. URL: https://en.wikipedia.org/w/index.php?title=Hadamard%20product%20(matrices).

[22] Wikipedia. Kronecker product, 2021. URL: https://en.wikipedia.org/w/index.php?title=Kronecker%20product.

[23] Wikipedia. Paley construction, 2021. URL: https://en.wikipedia.org/w/index.php?title=Paley%20construction.

[24] John Williamson. Hadamard' s determinant theorem and the sum of four squares. *Duke Mathematical Journal*, 11(1):65–81, 1944.

APPENDIX

Disclaim: the code will reports errors such as “declaration conflict”, if it is run with an up-to-date mathlib, as some bits have been merged into or are being merged to mathlib.

SET_FINSET_FINTYPE.LEAN

```
/-
Copyright (c) 2021 ***. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: ***.
-/

import tactic
import data.finset.basic
import data.fintype.card

/-!
This file supplements things about `set`, `finset`, `fintype`, that are current missing in mathlib.

## Main definition

- `set.card`: given a set `S : set I`, `S.card` is a shortcut of `S.to_finset.card`,
  under the conditions `[decidable_pred (λ (x : I), x ∈ S)] [fintype ↥S]`.
-/

local attribute [instance] set_fintype

open_locale big_operators

namespace set

variables {I : Type*} (S T U : set I)

lemma union_compl : S ∪ set.compl S = @set.univ I := by ext; simp

lemma union_compl' : S ∪ (λ x, ¬ (S x)) = @set.univ I :=
by {ext, simp, exact em (S x)}

@[simp] lemma inter_eq_empty_of_compl : S ∩ S.compl = ∅ := by simp

@[simp] lemma inter_eq_empty_of_compl' : S ∩ (λ x, ¬ (S x)) = ∅ := by {ext, simp, tauto}

lemma disjoint_of_compl: disjoint S S.compl := by simp [set.disjoint_iff_inter_eq_empty]

lemma disjoint_of_compl': disjoint S (λ x, ¬ (S x)) := by simp [set.disjoint_iff_inter_eq_empty]

/-- Given a set `S : set I`, `S.card` is a shortcut of `S.to_finset.card`. -/
def card [decidable_pred (λ (x : I), x ∈ S)] [fintype ↥S] : ℕ := S.to_finset.card

lemma univ_card_eq_fintype_card [fintype I] : (@set.univ I).card = fintype.card I :=
by simp [card, fintype.card]

@[simp] lemma coe_union_to_finset
[decidable_eq I] [fintype ↥S] [fintype ↥T] :
↑(S.to_finset ∪ T.to_finset) = S ∪ T :=
by ext; simp

instance union_decidable (x : I)
```

```
[decidable_pred (λ (x : I), x ∈ S)] [decidable_pred (λ (x : I), x ∈ T)] :
decidable (x ∈ (S ∪ T)) :=
by {simp [set.mem_union x S T], exact or.decidable}

instance union_decidable_pred
[decidable_pred (λ (x : I), x ∈ S)] [decidable_pred (λ (x : I), x ∈ T)] :
decidable_pred (λ (x : I), x ∈ (S ∪ T)) :=
infer_instance

variables {S T U}

lemma to_finset_union_eq_iff [decidable_eq I]
[fintype ↥S] [fintype ↥T] [fintype ↥U] :
S.to_finset ∪ T.to_finset = U.to_finset ↔ S ∪ T = U :=
by simp [←to_finset_union, set.to_finset_inj]

lemma card_disjoint_union [decidable_eq I]
[decidable_pred (λ (x : I), x ∈ S)] [fintype ↥S]
[decidable_pred (λ (x : I), x ∈ T)] [fintype ↥T]
(h : disjoint S T):
(S ∪ T).card = S.card + T.card :=
begin
  have h' := to_finset_disjoint_iff.2 h,
  dsimp [card],
  rw [← finset.card_disjoint_union h', to_finset_union],
end

lemma card_disjoint_union' [decidable_eq I]
[decidable_pred (λ (x : I), x ∈ S)] [fintype ↥S]
[decidable_pred (λ (x : I), x ∈ T)] [fintype ↥T]
[decidable_pred (λ (x : I), x ∈ U)] [fintype ↥U]
(d : disjoint S T) (u : S ∪ T = U) :
(U).card = S.card + T.card :=
begin
  rw ← card_disjoint_union d,
  congr,
  rw u,
end

end set

variables {α β I : Type*} [comm_monoid β] [fintype I]

open fintype finset

lemma finset.univ_eq_set_univ_to_finset : -- DO NOT use in simp!!!
finset.univ = (@set.univ I).to_finset := set.to_finset_univ.symm

lemma fintype.card_eq_finset_card_of_set (S : set α)
[fintype ↥S] [decidable_pred (λ (x : α), x ∈ S)]:
fintype.card S = finset.card (set.to_finset S) :=
by simp only [set.to_finset_card]

variable (I)
lemma fintype.card_eq_finset_card_of_univ :
fintype.card I = finset.card (@finset.univ I _):=
by simp only [fintype.card]
```

```
lemma fintype.card_eq_set_card_of_univ :
fintype.card I = (@set.univ I).card :=
by simp [set.card, fintype.card]

variable {I}

lemma finset.card_eq_sum_ones_ℚ {s : finset α}: (s.card : ℚ) = ∑ _ in s, 1 :=
by rw (finset.card_eq_sum_ones s); simp

@[to_additive]
lemma finset.prod_union' [decidable_eq α] {s₁ s₂ s : finset α} {f : α → β}
(d : disjoint s₁ s₂) (u : s₁ ∪ s₂ = s):
(∏ x in s, f x) = (∏ x in s₁, f x) * (∏ x in s₂, f x) :=
by simp [*, ← finset.prod_union]

@[to_additive]
lemma set.prod_union' [decidable_eq α] {S T U : set α} {f : α → β}
[fintype ↥S] [fintype ↥T] [fintype ↥U]
(d : disjoint S T) (u : S ∪ T = U):
(∏ x in U.to_finset, f x) =
(∏ x in S.to_finset, f x) *
(∏ x in T.to_finset, f x) :=
begin
  have d' := set.to_finset_disjoint_iff.2 d,
  have u' := set.to_finset_union_eq_iff.2 u,
  rw ← finset.prod_union' d' u',
end

lemma finset.card_erase_of_mem' [decidable_eq α] {a : α} {s : finset α} :
  a ∈ s →  finset.card s = finset.card (finset.erase s a) + 1:=
begin
  intro ha,
  have h:= finset.card_pos.mpr ⟨a, ha⟩,
  simp [finset.card_erase_of_mem ha, *],
  exact (nat.succ_pred_eq_of_pos h).symm
end

attribute [to_additive] fintype.prod_dite

lemma fintype.sum_split {α} {β} [fintype α] [add_comm_monoid β]
{f : α → β} (p : α → Prop) [decidable_pred p] :
  ∑ j, f j =
  ∑ j : {j : α // p j}, f j +
  ∑ j : {j : α //¬ p j}, f j :=
by simp [←fintype.sum_dite (λ a _, f a) (λ a _, f a)]

lemma fintype.sum_split' {α} {β} [fintype α] [add_comm_monoid β]
{f : α → β} (p q : α → Prop) [decidable_pred p] [decidable_pred q] :
  ∑ j : {j : α // p j}, f j =
  ∑ j : {j : α // p j ∧ q j}, f j +
  ∑ j : {j : α // p j ∧ ¬ q j}, f j :=
begin
  set q': (subtype p) → Prop := λ a, q (a.1),
  simp [fintype.sum_split q'],
  suffices h₁ : ∑ (j : {j // q' j}), f j = ∑ (j : {j // p j ∧ q j}), f j,
  suffices h₂ : ∑ (j : {j // ¬q' j}), f j = ∑ (j : {j // p j ∧ ¬q j}), f j,
  simp [←h₁, ←h₂],
  set g : {j // ¬q' j} → {j // p j ∧ ¬q j} := λ a, ⟨a.1.1, by tidy⟩,
```

```
  swap 2,
  set g : {j // q' j} → {j // p j ∧ q j} := λ a, ⟨a.1.1, by tidy⟩,
  any_goals
  { have hg : function.bijective g :=
      ⟨λ a b hab, by tidy, λ a, by tidy⟩,
    convert function.bijective.sum_comp hg _,
    ext, congr' 1 },
end

lemma fintype.card_split {α} [fintype α]
(p : α → Prop) [decidable_pred p] :
  fintype.card α =
  fintype.card {j : α // p j} +
  fintype.card {j : α // ¬ p j} :=
by simp only [fintype.card_eq_sum_ones, fintype.sum_split p]

lemma fintype.card_split' {α} [fintype α]
(p q : α → Prop) [decidable_pred p] [decidable_pred q]:
  fintype.card {j : α // p j} =
  fintype.card {j : α // p j ∧ q j} +
  fintype.card {j : α // p j ∧ ¬ q j} :=
begin
  have eq:= @fintype.sum_split' _ _ _ _ (λ _, 1) p q _ _,
  simp[*, fintype.card_eq_sum_ones] at *,
end

lemma finset.sum_split {α} {β} [add_comm_monoid β]
(s : finset α) {f : α → β} (p : α → Prop) [decidable_pred p] :
  ∑ j in s, f j =
  ∑ j in filter p s, f j +
  ∑ j in filter (λ (x : α), ¬p x) s, f j :=
by simp [←finset.sum_ite (λ j, f j) (λ j, f j)]

@[simp]
lemma finset.sum_filter_one {β} [add_comm_monoid β] [decidable_eq I]
(i : I) {f : I → β}:
∑ (x : I) in filter (λ (x : I), x = i) univ, f x = f i :=
begin
  simp [finset.filter_eq'],
end

@[simp]
lemma finset.sum_filter_two {β} [add_comm_monoid β] [decidable_eq I]
{i j : I} (h : i ≠ j) {f : I → β}:
∑ (k : I) in filter (λ (k : I), k = i ∨ k = j) univ, f k = f i + f j  :=
begin
  rw [finset.sum_split _ (λ k, k = i)],
  simp [finset.filter_eq', finset.filter_ne'],
  have : ∑ (x : I) in (filter (λ (k : I), k = i ∨ k = j) univ).erase i, f x
       = ∑ (x : I) in filter (λ (x : I), x = j) univ, f x,
  { apply finset.sum_congr,
    { ext, simp, split,
      { rintros ⟨h₁, (h₂ | h₂)⟩, contradiction, assumption },
      { rintros rfl, use ⟨h.symm, or.inr rfl⟩ } },
    { rintros, refl } },
  simp [this],
end
```

```
import group_theory.subgroup

/-!
This file supplements two instances relavant to monoid homomorphisms
-/

namespace monoid_hom
variables {G N: Type*} [group G] [group N]

/-- If the equality in `N` is decidable and `f : G →* N` is a `monoid_hom`,
    then the membership of `f.ker.carrier` is decidable. -/
instance [decidable_eq N] (f : G →* N) (x : G) :
decidable (x ∈ f.ker.carrier) := f.decidable_mem_ker x

/-- If `G` is a finite type, and the equality in `N` is decidable,
    and `f : G →* N` is a `monoid_hom`, then `f.ker.carrier` is a finite type. -/
instance [fintype G] [decidable_eq N] (f : G →* N) :
fintype (f.ker.carrier) := set_fintype (f.ker.carrier)

end monoid_hom
```

MATRIX_BASIC.LEAN

```
/-
Copyright (c) 2021 ***. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: ***.
-/
import linear_algebra.matrix.trace
import linear_algebra.matrix.to_lin
import linear_algebra.matrix.nonsingular_inverse
import data.complex.basic

/-!
This file supplements things about `matrix`, that are current missing in mathlib,
in particular those that will be used for the implementation of the Kronecker product
and the constructions of Hadamard matrices.

## Main definitions

Here only lists main definitions that are used for the implementation of
the Kronecker product and Hadamard matrices.
- `matrix.all_one`: the matrix whose entries are all `1`s.
- `matrix.is_sym`: a Proposition. Matrix `A` is symmetric `A.is_sym` means `Aᵀ = A`.
-/

variables {α β γ I J K L M N: Type*}
variables {R : Type*}
variables [fintype I] [fintype J] [fintype K] [fintype L] [fintype M] [fintype N]

/- ## reindex and coercion -/
instance prod_assoc :
has_coe ((α × β) × γ) (α × β × γ) :=
⟨λ ⟨⟨a,b⟩,c⟩, ⟨a,b,c⟩⟩
instance matrix.prod_assoc :
has_coe (matrix (I × J × K) (L × M × N) α) (matrix ((I × J) × K) ((L × M) × N) α):=
⟨λ M ⟨⟨a,b⟩,c⟩ ⟨⟨d,e⟩,f⟩, M ⟨a,b,c⟩ ⟨d,e,f⟩⟩
/-- `matrix.reindex_prod_assoc` constructs a natural equivalence between
```

```
    `matrix ((I × J) × K) ((L × M) × N) α` and `matrix (I × J × K) (L × M × N) α`. -/
def matrix.reindex_prod_assoc :
matrix ((I × J) × K) ((L × M) × N) α ≃ matrix (I × J × K) (L × M × N) α :=
matrix.reindex (equiv.prod_assoc _ _ _) (equiv.prod_assoc _ _ _)
/-- `matrix.reindex_prod_comm_fst` constructs a natural equivalence between
    `matrix I (J × K) α` and `matrix I (K × J) α`. -/
def matrix.reindex_prod_comm_fst :
matrix I (J × K) α ≃ matrix I (K × J) α :=
matrix.reindex (equiv.refl _) (equiv.prod_comm _ _)
/-- `matrix.reindex_prod_comm_snd` constructs a natural equivalence between
    `matrix (I × J) K α` and `matrix (J × I) K α`. -/
def matrix.reindex_prod_comm_snd :
matrix (I × J) K α ≃ matrix (J × I) K α :=
matrix.reindex (equiv.prod_comm _ _) (equiv.refl _)
/-- `matrix.reindex_prod_comm` constructs a natural equivalence between
    `matrix (I × J) (K × L) α` and `matrix (J × I) (L × K) α`. -/
def matrix.reindex_prod_comm :
matrix (I × J) (K × L) α ≃ matrix (J × I) (L × K) α :=
matrix.reindex (equiv.prod_comm _ _) (equiv.prod_comm _ _)
/- ## end reindex and coercion -/

/- ## perm matrix -/
/-- `equiv.perm.to_matrix σ` is the permutation matrix given by
    a permutation `σ : equiv.perm I` on the index tpye `I`. -/
def equiv.perm.to_matrix
[decidable_eq I] (α) [has_zero α] [has_one α] (σ : equiv.perm I) :
matrix I I α
| i j := if σ i = j then 1 else 0

/-- Proves `(σ.to_pequiv.to_matrix : matrix I I α)= σ.to_matrix α`. -/
lemma equiv.perm.to_matrix_eq_to_prequiv_to_matrix
[decidable_eq I] [has_zero α] [has_one α] (σ : equiv.perm I) :
(σ.to_pequiv.to_matrix : matrix I I α)= σ.to_matrix α:=
by ext i j; simp [pequiv.to_matrix, equiv.perm.to_matrix,equiv.to_pequiv]
/- ## end perm matrix -/
--------------------------------------------------------------------------------

namespace matrix
open_locale matrix big_operators
open complex

/- ## one -/
section one
variables [mul_one_class α] [add_comm_monoid α] [non_assoc_semiring β]

open_locale big_operators

@[simp] lemma dot_product_one (v : I → α) : dot_product v 1 = ∑ i, v i :=
by simp [dot_product]

@[simp] lemma dot_product_one' (v : I → α) :
dot_product v (λ i, 1) = ∑ i, v i :=
by simp [dot_product]

@[simp] lemma one_dot_product (v : I → α) :
dot_product 1 v = ∑ i, v i :=
by simp [dot_product]
```

```
@[simp] lemma one_dot_product' (v : I → α) :
dot_product (λ i, 1 : I → α) v = ∑ i, v i :=
by simp [dot_product]

lemma one_dot_one_eq_card :
dot_product (1 : I → α) 1 = fintype.card I :=
by simp [dot_product, fintype.card]

lemma one_dot_one_eq_card' :
dot_product (λ i, 1 : I → α) (λ i, 1) = fintype.card I :=
by simp [dot_product, fintype.card]

@[simp] lemma mul_vector_one (A : matrix I J β) :
mul_vec A 1 = λ i, ∑ j, A i j :=
by ext; simp [mul_vec, dot_product]

@[simp] lemma mul_vector_one' (A : matrix I J β) :
mul_vec A (λ i, 1) = λ i, ∑ j, A i j :=
by ext; simp [mul_vec, dot_product]

@[simp] lemma vector_one_mul (A : matrix I J β) :
vec_mul 1 A = λ j, ∑ i, A i j :=
by ext; simp [vec_mul, dot_product]

@[simp] lemma vector_one_mul' (A : matrix I J β) :
vec_mul (λ j, 1 : I → β) A = λ j, ∑ i, A i j :=
by ext; simp [vec_mul, dot_product]

end one

section all_one_matrix

/-- `matrix.all_one` is the matrix whose entries are all `1`s. -/
def all_one [has_one α]: matrix I J α := λ i, 1

localized "notation 𝟙 := matrix.all_one" in matrix

/-- `matrix.row sum A i` is the sum of entries of the row indexed by `i` of matrix `A`. -/
def row_sum [add_comm_monoid α] (A : matrix I J α) (i : I) := ∑ j, A i j

/-- `matrix.col_sum A j` is the sum of entries of the column indexed by `j` of matrix `A`. -/
def col_sum [add_comm_monoid α] (A : matrix I J α) (j : J) := ∑ i, A i j

lemma col_one_mul_row_one [non_assoc_semiring α] :
col (1 : I → α) ⬝ row (1 : I → α) = 𝟙 :=
by ext; simp [matrix.mul, all_one]

lemma row_one_mul_col_one [non_assoc_semiring α] :
row (1 : I → α) ⬝ col (1 : I → α) = fintype.card I • 1 :=
by {ext, simp [mul_apply, one_apply], congr,}

end all_one_matrix

/- ## end one -/

/- ## trace section -/
section trace
/-- The "trace" of a matrix.
```

```
    A different version of "trace" is defined in `trace.lean` as `matrix.trace`.
    One advantage of `matrix.tr` is that this definition is more elementary,
    which only requires α to be a `add_comm_monoid`; whilst `matrix.trace` requires
    `[semiring β] [add_comm_monoid α] [module β α]`.
    The equivalence can be easily established when `α` is indeed a `β-module`.
    Another advantage is that `matrix.tr` is more convenient for users to explore lemmas/theorems
    involving "trace" from a combinatorial aspect.-/
def tr [add_comm_monoid α] (A : matrix I I α) : α := ∑ i : I, A i i

/-- Establishes that `matrix.trace` is equivalent to `matrix.tr`. -/
lemma trace_eq_tr [semiring β] [add_comm_monoid α] [module β α] (A : matrix I I α)
: trace I β α A = tr A := rfl
end trace
/- ## end trace -/

/- ## conjugate transpose and symmetric -/
section conjugate_transpose

@[simp] lemma star_vec [has_star α] (v : I → α) (i : I) :
star v i = star (v i) := rfl

lemma trans_col_eq_row (A : matrix I J α) (i : I) : (λ j, Aᵀ j i) = A i :=
by simp [transpose]

lemma trans_row_eq_col (A : matrix I J α) (j : J) : Aᵀ j = (λ i, A i j):=
by ext; simp [transpose]

lemma eq_of_transpose_eq {A B : matrix I J α} : Aᵀ = Bᵀ → A = B :=
begin
  intros h, ext i j,
  have h':= congr_fun (congr_fun h j) i,
  simp [transpose] at h',
  assumption
end

/-- The conjugate transpose of a matrix defined in term of `star`. -/
def conj_transpose [has_star α] (M : matrix I J α) : matrix J I α
| x y := star (M y x)

localized "postfix `ᴴ`:1500 := matrix.conj_transpose" in matrix

@[simp] lemma conj_transpose_apply [has_star α] (M : matrix m n α) (i j) :
  M.conj_transpose j i = star (M i j) := rfl

@[simp] lemma conj_transpose_conj_transpose [has_involutive_star α] (M : matrix m n α) :
  Mᴴᴴ = M :=
by ext; simp

@[simp] lemma conj_transpose_zero [semiring α] [star_ring α] : (0 : matrix m n α)ᴴ = 0 :=
by ext i j; simp

@[simp] lemma conj_transpose_one [decidable_eq n] [semiring α] [star_ring α]:
  (1 : matrix n n α)ᴴ = 1 :=
by simp [conj_transpose]

@[simp] lemma conj_transpose_add
[semiring α] [star_ring α] (M : matrix m n α) (N : matrix m n α) :
  (M + N)ᴴ = Mᴴ + Nᴴ  := by ext i j; simp
```

```
@[simp] lemma conj_transpose_sub [ring α] [star_ring α] (M : matrix m n α) (N : matrix m n α) :
  (M - N)ᴴ = Mᴴ - Nᴴ  := by ext i j; simp

@[simp] lemma conj_transpose_smul [comm_monoid α] [star_monoid α] (c : α) (M : matrix m n α) :
  (c • M)ᴴ = (star c) • Mᴴ :=
by ext i j; simp [mul_comm]

@[simp] lemma conj_transpose_mul [semiring α] [star_ring α] (M : matrix m n α) (N : matrix n l α) :
  (M ⬝ N)ᴴ = Nᴴ ⬝ Mᴴ  := by ext i j; simp [mul_apply]

@[simp] lemma conj_transpose_neg [ring α] [star_ring α] (M : matrix m n α) :
  (- M)ᴴ = - Mᴴ  := by ext i j; simp

section star

/-- When `α` has a star operation, square matrices `matrix n n α` have a star
operation equal to `matrix.conj_transpose`. -/
instance [has_star α] : has_star (matrix n n α) := {star := conj_transpose}

lemma star_eq_conj_transpose [has_star α] (M : matrix m m α) : star M = Mᴴ := rfl

@[simp] lemma star_apply [has_star α] (M : matrix n n α) (i j) :
  (star M) i j = star (M j i) := rfl

instance [has_involutive_star α] : has_involutive_star (matrix n n α) :=
{ star_involutive := conj_transpose_conj_transpose }

/-- When `α` is a `*`-(semi)ring, `matrix.has_star` is also a `*`-(semi)ring. -/
instance [decidable_eq n] [semiring α] [star_ring α] : star_ring (matrix n n α) :=
{ star_add := conj_transpose_add,
  star_mul := conj_transpose_mul, }

/-- A version of `star_mul` for `⬝` instead of `*`. -/
lemma star_mul [semiring α] [star_ring α] (M N : matrix n n α) :
  star (M ⬝ N) = star N ⬝ star M := conj_transpose_mul _ _

end star

@[simp] lemma conj_transpose_minor
  [has_star α] (A : matrix m n α) (r_reindex : l → m) (c_reindex : o → n) :
  (A.minor r_reindex c_reindex)ᴴ = Aᴴ.minor c_reindex r_reindex :=
ext $ λ _ _, rfl

lemma conj_transpose_reindex [has_star α] (eₘ : m ≃ l) (eₙ : n ≃ o) (M : matrix m n α) :
  (reindex eₘ eₙ M)ᴴ = (reindex eₙ eₘ Mᴴ) :=
rfl

/-- Proposition `matrix.is_sym`. `A.is_sym` means `Aᵀ = A`. -/
def is_sym (A : matrix I I α) : Prop := Aᵀ = A

/-- Proposition `matrix.is_skewsym`. `A.is_skewsym` means `-Aᵀ = A` if `[has_neg α]`. -/
def is_skewsym [has_neg α] (A : matrix I I α) : Prop := -Aᵀ = A

/-- Proposition `matrix.is_Hermitian`. `A.is_Hermitian` means `Aᴴ = A` if `[has_star α]`. -/
def is_Hermitian [has_star α] (A : matrix I I α) : Prop := Aᴴ = A

end conjugate_transpose
/- ## end conjugate transpose and symmetric-/
```

```
/- ## definite section -/
section definite
open_locale complex_order

/-- Proposition `matrix.is_pos_def`. -/
def is_pos_def (M : matrix I I ℂ):=
M.is_Hermitian ∧ ∀ v : I → ℂ, v ≠ 0 → 0 < dot_product v (M.mul_vec v)

/-- Proposition `matrix.is_pos_semidef`. -/
def is_pos_semidef (M : matrix I I ℂ):= M
.is_Hermitian∧ ∀ v : I → ℂ, 0 ≤ dot_product v (M.mul_vec v)

/-- Proposition `matrix.is_neg_def`. -/
def is_neg_def (M : matrix I I ℂ):=
M.is_Hermitian ∧ ∀ v : I → ℂ, v ≠ 0 → dot_product v (M.mul_vec v) < 0

/-- Proposition `matrix.is_neg_semidef`. -/
def is_neg_semidef (M : matrix I I ℂ):=
M.is_Hermitian ∧ ∀ v : I → ℂ, dot_product v (M.mul_vec v) ≤ 0

end definite
/- ## end definite -/

/- ## matrix rank section  -/
section rank
variables [decidable_eq J] [field α]
/-- `rank A` is the rank of matrix `A`, defined as the rank of `A.to_lin'`. -/
noncomputable def rank (A : matrix I J α) := rank A.to_lin'
end rank
/- ## end matrix rank -/

/- ## orthogonal section  -/
section orthogonal
variable [decidable_eq I]

/-- Proposition `matrix.is_ortho`.
    `A` is orthogonal `A.is_ortho` means `Aᵀ · A = 1`, where `A : matrix I I ℝ`. -/
def is_ortho (A : matrix I I ℝ) : Prop := Aᵀ · A = 1

/-- Proposition `matrix.is_uni`.
    `A` is unitray `A.is_uni` means `A^H · A = 1`, where `A : matrix I I ℂ`. -/
def is_uni (A : matrix I I ℂ) : Prop := Aᴴ · A = 1

lemma is_ortho_left_right (A : matrix I I ℝ) :
A.is_ortho ↔ A · Aᵀ = 1 :=
⟨nonsing_inv_right_left, nonsing_inv_left_right⟩

lemma is_uni_left_right (A : matrix I I ℂ) :
A.is_uni ↔ A · Aᴴ = 1 :=
⟨nonsing_inv_right_left, nonsing_inv_left_right⟩

/-- A matrix `A` is orthogonal iff `A` has orthonormal columns. -/
lemma is_ortho_iff_orthonormal_cols (A : matrix I I ℝ) :
matrix.is_ortho A ↔ ∀ j₁ j₂, dot_product (λ i, A i j₁) (λ i, A i j₂) = ite (j₁ = j₂) 1 0 :=
begin
  simp [matrix.is_ortho,matrix.mul,has_one.one, diagonal],
  split,
  { intros h j₁ j₂,
```

```
    exact congr_fun (congr_fun h j₁) j₂,
  },
  { intros h, ext, apply h _ _},
end

/-- A matrix `A` is orthogonal iff `A` has orthonormal rows. -/
lemma is_ortho_iff_orthonormal_row (A : matrix I I ℝ) :
matrix.is_ortho A ↔ ∀ i₁ i₂, dot_product (A i₁) (A i₂) = ite (i₁ = i₂) 1 0 :=
begin
  rw is_ortho_left_right,
  simp [matrix.is_ortho,matrix.mul,has_one.one, diagonal],
  split,
  { intros h i₁ i₂,
    exact congr_fun (congr_fun h i₁) i₂,
  },
  { intros h, ext, apply h _ _},
end

end orthogonal
/- ## end orthogonal -/

/- ## permutation matrix -/
section perm
open equiv

variables [decidable_eq I] [has_zero α] [has_one α]

/-- `P.is_perm` if matrix `P` is induced by some permutation `σ`. -/
def is_perm (P : matrix I I α) : Prop :=
∃ σ : equiv.perm I, P = perm.to_matrix α σ

/-- `P.is_perfect_shuffle` if matrix `P` is induced by some permutation `σ`, and `∀ i : I, σ i ≠ i`. -/
def is_perfect_shuffle (P : matrix I I α) : Prop :=
∃ σ : equiv.perm I, (P = perm.to_matrix α σ ∧ ∀ i : I, σ i ≠ i)

/-- A permutation matrix is a perfect shuffle. -/
lemma is_perm_of_is_perfect_shuffle (P : matrix I I α) :
P.is_perfect_shuffle → P.is_perm :=
by {intro h, rcases h with ⟨σ, rfl, h2⟩, use σ}

end perm
/- ## end permutation -/


/- ## matrix similarity section  -/
section similarity
variables [comm_ring α] [decidable_eq I]

/-- `matrix.similar_to` defines the proposition that
    matrix `A` is similar to `B`, denoted as `A ~ B`. -/
def similar_to (A B : matrix I I α) :=
∃ (P : matrix I I α), is_unit P.det ∧ B = P⁻¹ * A * P
localized "notation `~`:50 := similar_to" in matrix

/-- An equivalent definition of matrix similarity `similar_to`. -/
def similar_to' (A B : matrix I I α) :=
∃ (P : matrix I I α), is_unit P ∧ B = P⁻¹ * A * P
```

```
/-- `matrix.perm_similar_to` defines the proposition that
    matrix `A` is permutation-similar to `B`, denoted as `A ~□ B`.-/
def perm_similar_to (A B : matrix I I α) :=
∃ (P : matrix I I α), P.is_perm ∧ B = P⁻¹ · A · P
localized "notation `~□`:50 := perm_similar_to" in matrix

/-- Proves the equivalence of `matrix.similar_to` and `matrix.similar_to'`. -/
lemma similar_to_iff_similar_to' (A B : matrix I I α) :
similar_to A B ↔ similar_to' A B :=
⟨ by {rintros ⟨P ,h1, h2⟩, rw ←is_unit_iff_is_unit_det at h1, use⟨P ,h1, h2⟩},
  by {rintros ⟨P ,h1, h2⟩, rw is_unit_iff_is_unit_det at h1, use⟨P ,h1, h2⟩} ⟩

end similarity
/- ## end matrix similarity -/

/- ## others -/
/-- Two empties matrices are equal. -/
lemma eq_of_empty [c: is_empty I] (M N: matrix I I α) : M = N :=
by {ext, exfalso, apply is_empty_iff.mp c i}

/-- `matrix.dot_product_block'` splits the `dot_product` of
    two block vectors into a sum of two `Σ` sums. -/
lemma dot_product_block' [has_mul α] [add_comm_monoid α] (v w : I ⊕ J → α) :
dot_product v w =
∑ i, v (sum.inl i) * w (sum.inl i) +
∑ j, v (sum.inr j) * w (sum.inr j) :=
begin
  rw [dot_product, ←fintype.sum_sum_elim],
  congr,
  ext (i | j); simp
end

/-- `matrix.dot_product_block` splits the `dot_product` of
    two block vectors into a sum of two `dot_product` . -/
lemma dot_product_block [has_mul α] [add_comm_monoid α] (v w : I ⊕ J → α) :
dot_product v w =
dot_product (λ i, v (sum.inl i))  (λ i, w (sum.inl i)) +
dot_product (λ j, v (sum.inr j))  (λ j, w (sum.inr j)) :=
by simp [dot_product, dot_product_block']
/- ## end others -/

end matrix
```

INV_MATRIX.LEAN

```
/-
Copyright (c) 2021 ***. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: ***.
-/

import linear_algebra.matrix.nonsingular_inverse

/-!
This file supplements things about matrix inverse, that were missing in mathlib.
-/

namespace matrix
```

```
universes u v
variables {n : Type u} [decidable_eq n] [fintype n] {α : Type v} [comm_ring α]
open_locale matrix big_operators
open equiv equiv.perm finset

variables (A : matrix n n α) (B : matrix n n α)

/- `is_unit_of_invertible A`
   converts the "stronger" condition `invertible A` to proposition `is_unit A`. -/

/-- `matrix.is_unit_det_of_invertible` converts `invertible A` to `is_unit A.det`. -/
lemma is_unit_det_of_invertible [invertible A] : is_unit A.det :=
@is_unit_of_invertible _ _ _(det_invertible_of_invertible A)

@[simp]
lemma inv_eq_nonsing_inv_of_invertible [invertible A] : ⅟ A = A⁻¹ :=
begin
  suffices : is_unit A,
  { rw [←this.mul_left_inj, inv_of_mul_self, matrix.mul_eq_mul, nonsing_inv_mul],
    rwa ←is_unit_iff_is_unit_det },
  exact is_unit_of_invertible _
end

variables {A} {B}

/- `is_unit.invertible` lifts the proposition `is_unit A` to a constructive inverse of `A`. -/

/-- "Lift" the proposition `is_unit A.det` to a constructive inverse of `A`. -/
noncomputable def invertible_of_is_unit_det  (h : is_unit A.det) : invertible A :=
⟨A⁻¹, nonsing_inv_mul A h, mul_nonsing_inv A h⟩

/-- If matrix A is left invertible, then its inverse equals its left inverse. -/
lemma inv_eq_left_inv (h : B ⬝ A = 1) : A⁻¹ = B :=
begin
  have h1 :=  (is_unit_det_of_left_inverse h),
  have h2 := matrix.invertible_of_is_unit_det h1,
  have := @inv_of_eq_left_inv (matrix n n α) (infer_instance) A B h2 h,
  simp* at *,
end

/-- If matrix A is right invertible, then its inverse equals its right inverse. -/
lemma inv_eq_right_inv (h : A ⬝ B = 1) : A⁻¹ = B :=
begin
  have h1 :=  (is_unit_det_of_right_inverse h),
  have h2 := matrix.invertible_of_is_unit_det h1,
  have := @inv_of_eq_right_inv (matrix n n α) (infer_instance) A B h2 h,
  simp* at *,
end

/-- We can construct an instance of invertible A if A has a left inverse. -/
def invertible_of_left_inverse (h: B ⬝ A = 1) : invertible A :=
⟨B, h, nonsing_inv_right_left h⟩

/-- We can construct an instance of invertible A if A has a right inverse. -/
def invertible_of_right_inverse (h: A ⬝ B = 1) : invertible A :=
⟨B, nonsing_inv_left_right h, h⟩

variables {C: matrix n n α}
```

```
/-- The left inverse of matrix A is unique when existing. -/
lemma left_inv_eq_left_inv (h: B · A = 1) (g: C · A = 1) : B = C :=
by rw [←(inv_eq_left_inv h), ←(inv_eq_left_inv g)]

/-- The right inverse of matrix A is unique when existing. -/
lemma right_inv_eq_right_inv (h: A · B = 1) (g: A · C = 1) : B = C :=
by rw [←(inv_eq_right_inv h), ←(inv_eq_right_inv g)]

/-- The right inverse of matrix A equals the left inverse of A when they exist. -/
lemma right_inv_eq_left_inv (h: A · B = 1) (g: C · A = 1) : B = C :=
by rw [←(inv_eq_right_inv h), ←(inv_eq_left_inv g)]

variable (A)

@[simp] lemma mul_inv_of_invertible [invertible A] : A · A⁻¹ = 1 :=
mul_nonsing_inv A (is_unit_det_of_invertible A)

@[simp] lemma inv_mul_of_invertible [invertible A] : A⁻¹ · A = 1 :=
nonsing_inv_mul A (is_unit_det_of_invertible A)

end matrix
```

SYMMETRIC_MATRIX.LEAN

```
/-
Copyright (c) 2021 ***. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: ***.
-/
import set_finset_fintype
import matrix_basic

/-!
# Symmetric matrices

This file contains basic results about symmetric matrices.
See `matrix_basic.lean` for the definition of symmetric matrices.

## Tags

sym, symmetric, matrix
-/

namespace matrix

open_locale matrix

variables {α I J R : Type*} [fintype I] [fintype J]

lemma is_sym.eq {A : matrix I I α} (h : A.is_sym) : Aᵀ = A := h

lemma is_sym.ext_iff {A : matrix I I α} : A.is_sym ↔ ∀ i j, Aᵀ i j = A i j :=
by rw [is_sym, matrix.ext_iff]

lemma is_sym.ext_iff' {A : matrix I I α} : A.is_sym ↔ ∀ i j, A j i = A i j :=
is_sym.ext_iff

lemma is_sym.apply {A : matrix I I α} (h : A.is_sym) (i j : I) : Aᵀ i j = A i j :=
```

```
is_sym.ext_iff.1 h i j

lemma is_sym.apply' {A : matrix I I α} (h : A.is_sym) (i j : I) : A j i = A i j :=
is_sym.apply h i j

lemma is_sym.ext {A : matrix I I α} : (∀ i j, Aᵀ i j = A i j) → A.is_sym :=
is_sym.ext_iff.2

lemma is_sym.ext' {A : matrix I I α} : (∀ i j, Aᵀ i j = A i j) → A.is_sym :=
is_sym.ext

/-- A block matrix `A.from_blocks B C D` is symmetric, if `A` and `D` are symmetric and `Bᵀ = C`. -/
lemma is_sym_of_block_conditions
{A : matrix I I α} {B : matrix I J α} {C : matrix J I α} {D : matrix J J α} :
(A.is_sym) ∧ (D.is_sym) ∧ (Bᵀ = C) → (A.from_blocks B C D).is_sym :=
begin
  rintros ⟨h1, h2, h3⟩,
  have h4 : Cᵀ = B, {rw ← h3, simp},
  unfold matrix.is_sym,
  rw from_blocks_transpose,
  congr;
  assumption
end

/-- `A · Aᵀ` is symmertric. -/
lemma mul_transpose_self_is_sym [comm_semiring α] (A : matrix I I α) :
(A · Aᵀ).is_sym :=
by simp [matrix.is_sym, transpose_mul]

/-- The identity matrix is symmetric. -/
@[simp] lemma is_sym_of_one [decidable_eq I] [has_zero α] [has_one α] :
(1 : matrix I I α).is_sym := by {ext, simp}

/-- The negtive identity matrix is symmetric. -/
@[simp] lemma is_sym_of_neg_one [decidable_eq I] [has_zero α] [has_one α] [has_neg α] :
(-1 : matrix I I α).is_sym := by {ext, simp}

/-- The identity matrix multiplied by any scalar `k` is symmetric. -/
@[simp] lemma is_sym_of_smul_one
[decidable_eq I] [monoid R] [add_monoid α] [has_one α] [distrib_mul_action R α] (k : R) :
(k • (1 : matrix I I α)).is_sym :=
by { ext, simp [is_sym_of_one.apply'] }

/-- If a block matrix `A.from_blocks B C D` is symmetric,-/
lemma block_conditions_of_is_sym
{A : matrix I I α} {B : matrix I J α} {C : matrix J I α} {D : matrix J J α} :
(A.from_blocks B C D).is_sym → (A.is_sym) ∧ (D.is_sym) ∧ (Cᵀ = B) ∧ (Bᵀ = C) :=
begin
  rintros h,
  unfold matrix.is_sym at h,
  rw from_blocks_transpose at h,
  have h1 : (Aᵀ.from_blocks Cᵀ Bᵀ Dᵀ).to_blocks₁₁ = (A.from_blocks B C D).to_blocks₁₁, {rw h},
  have h2 : (Aᵀ.from_blocks Cᵀ Bᵀ Dᵀ).to_blocks₁₂ = (A.from_blocks B C D).to_blocks₁₂, {rw h},
  have h3 : (Aᵀ.from_blocks Cᵀ Bᵀ Dᵀ).to_blocks₂₁ = (A.from_blocks B C D).to_blocks₂₁, {rw h},
  have h4 : (Aᵀ.from_blocks Cᵀ Bᵀ Dᵀ).to_blocks₂₂ = (A.from_blocks B C D).to_blocks₂₂, {rw h},
  simp at *,
  use ⟨h1, h4, h2, h3⟩
end
```

```
end matrix
```

DIAGONAL_MATRIX.LEAN

```
/-
Copyright (c) 2021 ***. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: ***.
-/
import symmetric_matrix
import main1

/-!
# Diagonal matrices

This file contains the definition and basic results about diagonal matrices.

## Main results

- `matrix.has_orthogonal_rows`: `A.has_orthogonal_rows` means `A` has orthogonal
                                (with respect to `dot_product`) rows.
- `matrix.has_orthogonal_cols`: `A.has_orthogonal_cols` means `A` has orthogonal
                                (with respect to `dot_product`) columns.
- `matrix.is_diagonal`: a proposition that stats a given square matrix `A` is diagonal
                        (i.e. `∀ i j, i ≠ j → A i j = 0`).

## Tags

diag, diagonal, matrix
-/

namespace matrix

variables {α R I J : Type*} [fintype I] [fintype J]
open_locale matrix

/-- `A.has_orthogonal_rows` means matrix `A` has orthogonal (with respect to `dot_product`) rows. -/
def has_orthogonal_rows [has_mul α] [add_comm_monoid α] (A : matrix I J α) : Prop :=
∀ {i₁ i₂}, i₁ ≠ i₂ → dot_product (A i₁) (A i₂) = 0

/-- `A.has_orthogonal_cols` means matrix `A` has orthogonal (with respect to `dot_product`) columns. -/
def has_orthogonal_cols [has_mul α] [add_comm_monoid α] (A : matrix I J α) : Prop :=
∀ {i₁ i₂}, i₁ ≠ i₂ → dot_product (λ j, A  j i₁) (λ j, A j i₂) = 0

/-- `Aᵀ` has orthogonal rows iff `A` has orthogonal columns. -/
lemma transpose_has_orthogonal_rows_iff_has_orthogonal_cols
[has_mul α] [add_comm_monoid α] (A : matrix I J α) :
Aᵀ.has_orthogonal_rows ↔ A.has_orthogonal_cols :=
by simp [has_orthogonal_rows, has_orthogonal_cols, trans_row_eq_col]

/-- `Aᵀ` has orthogonal columns iff `A` has orthogonal rows. -/
lemma transpose_has_orthogonal_cols_iff_has_orthogonal_rows
[has_mul α] [add_comm_monoid α] (A : matrix I J α) :
Aᵀ.has_orthogonal_cols ↔ A.has_orthogonal_rows :=
by simp [has_orthogonal_rows, has_orthogonal_cols, trans_row_eq_col]

/-- `A.is_diagonal` means square matrix `A` is a dianogal matrix: `∀ i j, i ≠ j → A i j = 0`. -/
def is_diagonal [has_zero α] (A : matrix I I α) : Prop := ∀ i j, i ≠ j → A i j = 0
```

```
/-- Non-diagonal entries equal zero. -/
@[simp] lemma is_diagonal.apply_ne [has_zero α] {A : matrix I I α}
(ha : A.is_diagonal) {i j : I} (h : ¬ i = j) :
A i j = 0 := ha i j h

/-- Non-diagonal entries equal zero. -/
@[simp] lemma is_diagonal.apply_ne' [has_zero α] {A : matrix I I α}
(ha : A.is_diagonal) {i j : I} (h : ¬ j = i) :
A i j = 0 := ha.apply_ne (ne.symm h)

/-- Matrix `A` is diagonal iff there is a vector `d` such that A is the diagonal matix generated by `d`. -/
lemma is_diagonal_iff_eq_diagonal [has_zero α] [decidable_eq I] (A : matrix I I α) :
A.is_diagonal ↔ (∃ d, A = diagonal d):=
begin
  split,
  { intros h, use (λ i, A i i),
    ext,
    specialize h i j,
    by_cases i = j;
    simp * at *, },
  { rintros ⟨d, rfl⟩ i j h, simp *}
end

/-- Every unit matrix is diagonal. -/
@[simp] lemma is_diagonal_of_unit [has_zero α] (A : matrix unit unit α) : A.is_diagonal :=
by {intros i j h, have h':= @unit.ext i j, simp* at *}

/-- Every zero matrix is diagonal. -/
@[simp] lemma is_diagonal_of_zero [has_zero α] : (0 : matrix I I α).is_diagonal :=
λ i j h, by simp

/-- Every identity matrix is diagonal. -/
@[simp] lemma is_diagonal_of_one [decidable_eq I] [has_zero α] [has_one α] :
  (1 : matrix I I α).is_diagonal :=
by {intros i j h, simp *}

/-- `smul` identity matrix `1` is diagonal. -/
@[simp] lemma is_diagonal_of_smul_one
[decidable_eq I] [monoid R] [add_monoid α] [has_one α] [distrib_mul_action R α] (k : R) :
(k • (1 : matrix I I α)).is_diagonal := by {intros i j h, simp *}

/-- Matrix `k • A` is diagonal if `A` is. -/
@[simp] lemma smul_is_diagonal_of [monoid R] [add_monoid α] [distrib_mul_action R α]
{k : R} {A : matrix I I α} (ha : A.is_diagonal):
(k • A).is_diagonal := by {intros i j h, simp [ha i j h]}

/-- The sum of two diagonal matrices is diagonal. -/
@[simp] lemma is_diagonal_add [add_zero_class α] {A B : matrix I I α} (ha : A.is_diagonal)
  (hb : B.is_diagonal) : (A + B).is_diagonal :=
  by {intros i j h, simp *, rw [ha i j h, hb i j h], simp}

/-- The block matrix `A.from_blocks B C D` is diagonal if  `A` and `D` are diagonal and `B`
    and `C` are `0`. -/
lemma is_diagnoal_of_block_conditions [has_zero α]
{A : matrix I I α} {B : matrix I J α} {C : matrix J I α} {D : matrix J J α} :
(A.is_diagonal) ∧ (D.is_diagonal) ∧ (B = 0) ∧ (C = 0) → (A.from_blocks B C D).is_diagonal:=
begin
  rintros h (i | i) (j | j) hij,
```

```
  any_goals {rcases h with ⟨ha, hd, hb, hc⟩, simp* at *},
  {have h' : i ≠ j, {simp* at *}, exact ha i j h'},
  {have h' : i ≠ j, {simp* at *}, exact hd i j h'},
end

/-- A symmetric block matrix `A.from_blocks B C D` is diagonal if  `A` and `D` are diagonal
    and `B` is `0`. -/
lemma is_diagnoal_of_sym_block_conditions [has_zero α]
{A : matrix I I α} {B : matrix I J α} {C : matrix J I α} {D : matrix J J α}
(sym : (A.from_blocks B C D).is_sym) :
(A.is_diagonal) ∧ (D.is_diagonal) ∧ (B = 0) → (A.from_blocks B C D).is_diagonal:=
begin
  rintros h,
  apply is_diagnoal_of_block_conditions,
  refine ⟨h.1, h.2.1, h.2.2, _⟩,
  obtain ⟨g1, g2, g3, g4⟩ := block_conditions_of_is_sym sym,
  simp [← g4, h.2.2]
end

/-- A different form of `matrix.is_diagnoal_of_sym_block_conditions`. -/
lemma is_diagnoal_of_sym_block_conditions' [has_zero α]
{A : matrix I I α} {B : matrix I J α} {C : matrix J I α} {D : matrix J J α}
{M : matrix (I ⊕ J) (I ⊕ J) α} (sym : M.is_sym) (h : M = A.from_blocks B C D) :
(A.is_diagonal) ∧ (D.is_diagonal) ∧ (B = 0) → M.is_diagonal:=
by rw h at *; convert is_diagnoal_of_sym_block_conditions sym

/-- `(A · Aᵀ).is_diagonal` iff `A.has_orthogonal_rows`. -/
lemma mul_tranpose_is_diagonal_iff_has_orthogonal_rows
[has_mul α] [add_comm_monoid α] {A : matrix I J α} :
(A · Aᵀ).is_diagonal ↔ A.has_orthogonal_rows :=
begin
  split,
  { rintros h i1 i2 hi,
    have h' := h i1 i2 hi,
    simp [dot_product, mul_apply,*] at *, },
  { intros ha i j h,
    have h':= ha h,
    simp [mul_apply, *, dot_product] at *,
  }
end

/-- `(Aᵀ · A).is_diagonal` iff `A.has_orthogonal_cols`. -/
lemma tranpose_mul_is_diagonal_iff_has_orthogonal_cols
[has_mul α] [add_comm_monoid α] {A : matrix I J α} :
(Aᵀ · A).is_diagonal ↔ A.has_orthogonal_cols :=
begin
  simp [←transpose_has_orthogonal_rows_iff_has_orthogonal_cols],
  convert mul_tranpose_is_diagonal_iff_has_orthogonal_rows,
  simp
end

/-- `(A ⊗ B).is_diagonal` if both `A` and `B` are diagonal. -/
lemma K_is_diagonal_of [mul_zero_class α]
{A : matrix I I α} {B : matrix J J α} (ga: A.is_diagonal) (gb: B.is_diagonal):
(A ⊗ B).is_diagonal :=
begin
  rintros ⟨a, b⟩ ⟨c, d⟩ h,
  simp [Kronecker],
```

```
  by_cases ha: a = c,
  have hb: b ≠ d, {intros hb, rw [ha, hb] at h, apply h rfl},
  rw (gb _ _ hb), simp,
  rw (ga _ _ ha), simp,
end

end matrix
```

CIRCULANT_MATRIX.LEAN

```
/-
Copyright (c) 2021 ***. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: ***.
-/
import symmetric_matrix

/-!
# Circulant matrices

This file contains the definition and basic results about circulant matrices.

## Main results

- `matrix.cir`: introduce the definition of a circulant matrix generated by a given vector `v : I → α`.

## Implementation notes

`fin.foo` is the `fin n` version of `foo`.
Namely, the index type of the circulant matrices in discussion is `fin n`.

## Tags

cir, matrix
-/

variables {α I R : Type*} [fintype I] {n : ℕ}

namespace matrix
open_locale matrix big_operators

/-- Given the condition `[has_sub I]` and a vector `v : I → α`,
    we define `cir v` to be the circulant matrix generated by `v` of type `matrix I I α`. -/
def cir [has_sub I] (v : I → α) : matrix I I α
| i j := v (i - j)

/-- When `I` is an `add_group`, the 0th column of `cir v` is `v`. -/
lemma cir_col_zero_eq [add_group I] (v : I → α) :
(λ i, (cir v) i 0) = v := by ext; simp [cir]

/-- When `I` is an `add_group`, `cir v = cir w ↔ v = w`. -/
lemma cir_ext_iff [add_group I] {v w : I → α} :
cir v = cir w ↔ v = w :=
begin
  split,
  { intro h, rw [← cir_col_zero_eq v, ← cir_col_zero_eq w, h] },
  { rintro rfl, refl }
end
```

```
lemma fin.cir_ext_iff {v w : fin n → α} :
cir v = cir w ↔ v = w :=
begin
  induction n with n ih,
  {tidy},
  exact cir_ext_iff
end

/-- The sum of two circulant matrices `cir v` and `cir w` is also a circulant matrix `cir (v + w)`. -/
lemma cir_add [has_add α] [has_sub I] (v w : I → α) :
cir v + cir w = cir (v + w) := by ext; simp [cir]

/-- The product of two circulant matrices `cir v` and `cir w` is also a circulant matrix
    `cir (mul_vec (cir w) v)`. -/
lemma cir_mul [comm_semiring α] [add_comm_group I] (v w : I → α) :
cir v ⬝ cir w = cir (mul_vec (cir w) v) :=
begin
  ext i j,
  simp [mul_apply, mul_vec, cir, dot_product],
  refine fintype.sum_equiv ((equiv.add_left (-i)).trans (equiv.neg _)) _ _ _,
  simp [mul_comm],
  intro x,
  congr' 2; abel
end

lemma fin.cir_mul [comm_semiring α] (v w : fin n → α) :
cir v ⬝ cir w = cir (mul_vec (cir w) v) :=
begin
  induction n with n ih, {refl},
  exact cir_mul v w,
end

/-- Circulant matrices commute in multiplication under certain condations. -/
lemma cir_mul_comm
[comm_semigroup α] [add_comm_monoid α] [add_comm_group I] (v w : I → α) :
cir v ⬝ cir w = cir w ⬝ cir v :=
begin
  ext i j,
  simp [mul_apply, cir, mul_comm],
  refine fintype.sum_equiv (((equiv.add_right (-i)).trans (equiv.neg _)).trans (equiv.add_right j)) _ _ _,
  simp,
  intro x,
  congr' 2; abel
end

lemma fin.cir_mul_comm
[comm_semigroup α] [add_comm_monoid α] (v w : fin n → α) :
cir v ⬝ cir w = cir w ⬝ cir v :=
begin
  induction n with n ih, {refl},
  exact cir_mul_comm v w,
end

/-- `k • cir v` is another circluant matrix `cir (k • v)`. -/
lemma smul_cir [has_sub I] [has_scalar R α] {k : R} {v : I → α} :
k • cir v = cir (k • v) := by {ext, simp [cir]}

/-- The identity matrix is a circulant matrix. -/
```

```
lemma one_eq_cir [has_zero α] [has_one α] [decidable_eq I] [add_group I]:
(1 : matrix I I α) = cir (λ i, ite (i = 0) 1 0) :=
begin
  ext,
  simp [cir, one_apply],
  congr' 1,
  apply propext,
  exact sub_eq_zero.symm
end

/-- An alternative version of `one_eq_cir`. -/
lemma one_eq_cir' [has_zero α] [has_one α] [decidable_eq I] [add_group I]:
(1 : matrix I I α) = cir (λ i, (1 : matrix I I α) i 0) := one_eq_cir

lemma fin.one_eq_cir [has_zero α] [has_one α] :
(1 : matrix (fin n) (fin n) α) = cir (λ i, ite (i.1 = 0) 1 0) :=
begin
  induction n with n, {dec_trivial},
  convert one_eq_cir,
  ext, congr' 1,
  apply propext,
  exact (fin.ext_iff x 0).symm
end

/-- For a one-ary predicate `p`, `p` applied to every entry of `cir v` is true if `p`
    applied to every entry of `v` is true. -/
lemma pred_cir_entry_of_pred_vec_entry [has_sub I] {p : α → Prop} {v : I → α} :
(∀ k, p (v k)) → ∀ i j, p ((cir v) i j) :=
begin
  intros h i j,
  simp [cir],
  exact h (i - j),
end

/-- Given a set `S`, every entry of `cir v` is in `S` if every entry of `v` is in `S`. -/
lemma cir_entry_in_of_vec_entry_in [has_sub I] {S : set α} {v : I → α} :
(∀ k, v k ∈ S) → ∀ i j, (cir v) i j ∈ S :=
@pred_cir_entry_of_pred_vec_entry α I _ _ S v

/-- The circulant matrix `cir v` is symmetric iff `∀ i j, v (j - i) = v (i - j)`. -/
lemma cir_is_sym_ext_iff' [has_sub I] {v : I → α} :
(cir v).is_sym ↔ ∀ i j, v (j - i) = v (i - j) :=
by simp [is_sym.ext_iff, cir]

/-- The circulant matrix `cir v` is symmetric iff `v (- i) = v i` if `[add_group I]`. -/
lemma cir_is_sym_ext_iff [add_group I] {v : I → α} :
(cir v).is_sym ↔ ∀ i, v (- i) = v i :=
begin
  rw [cir_is_sym_ext_iff'],
  split,
  { intros h i, convert h i 0; simp },
  { intros h i j, convert h (i - j), simp }
end

lemma fin.cir_is_sym_ext_iff {v : fin n → α} :
(cir v).is_sym ↔ ∀ i, v (- i) = v i :=
begin
  induction n with n ih,
```

```
  { rw [cir_is_sym_ext_iff'],
    split;
    {intros h i, have :=i.2, simp* at *} },
  convert cir_is_sym_ext_iff,
end

/-- If `cir v` is symmetric, `∀ i j : I, v (j - i) = v (i - j)`. -/
lemma cir_is_sym_apply' [has_sub I] {v : I → α} (h : (cir v).is_sym) (i j : I) :
v (j - i) = v (i - j) := cir_is_sym_ext_iff'.1 h i j

/-- If `cir v` is symmetric, `∀ i j : I, v (- i) = v i`. -/
lemma cir_is_sym_apply [add_group I] {v : I → α} (h : (cir v).is_sym) (i : I) :
v (-i) = v i := cir_is_sym_ext_iff.1 h i

lemma fin.cir_is_sym_apply {v : fin n → α} (h : (cir v).is_sym) (i : fin n) :
v (-i) = v i := fin.cir_is_sym_ext_iff.1 h i

/-- The associated polynomial `(v 0) + (v 1) * X + ... + (v (n-1)) * X ^ (n-1)` to `cir v`.-/
noncomputable def cir_poly [semiring α] (v : fin n → α) : polynomial α :=
∑ i : fin n, polynomial.monomial i (v i)

/-- `cir_perm n` is the cyclic permutation over `fin n`. -/
def cir_perm : Π n, equiv.perm (fin n) := λ n, equiv.symm (fin_rotate n)

/-- `cir_P α n` is the cyclic permutation matrix of order `n` with entries of type `α`. -/
def cir_P (α) [has_zero α] [has_one α] (n : ℕ) :
matrix (fin n) (fin n) α := equiv.perm.to_matrix α (cir_perm n)

end matrix
```

FINITE_FIELD.LEAN

```
/-
Copyright (c) 2021 ***. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: ***.
-/
import field_theory.finite.basic
import group_theory.quotient_group
import monoid_hom
import set_finset_fintype

/-!
# Finite fields

This file supplements basic results about finite field that are missing in mathlib.
As well, this files defines quadratic residues and the quadratic character,
and contains basic results about them.
Throughout most of this file, `F` denotes a finite field,
and `q` is notation for the cardinality of `F`, and `p` is the character of `F`.

## Main results
1. `finite_field.is_quad_residue`: a proposition predicating whether a given `a : F` is a
                                   quadratic residue in `F`.
   `finite_field.is_quad_residue a` is defined to be `a ≠ 0 ∧ ∃ b, a = b^2`.
2. `finite_field.is_non_residue`: a proposition predicating whether a given `a : F` is a
                                  quadratic non-residue in `F`.
   `finite_field.is_non_residue a` is defined to be `a ≠ 0 ∧ ¬ ∃ b, a = b^2`.
```

```
3. `finite_field.quad_residues`: the subtype of `F` containing quadratic residues.
4. `finite_field.quad_residues_set`: the set containing quadratic residues of `F`.
5. `finite_field.non_residues`: the subtype of `F` containing quadratic non-residues.
6. `finite_field.non_residues_set`: the set containing quadratic non-residues of `F`.

7. `finite_field.sq`: the square function from `units F` to `units F`, defined as a group homomorphism.

8. `finite_filed.non_residue_mul`: is the map `_ * b` given a non-residue `b`
   defined on `{a : F // is_quad_residue a}`.

9. `finite_field.quad_char`: defines the quadratic character of `a` in a finite field `F`.
   - `χ a = 0 ` if `a = 0`
   - `χ a = 1` if `a ≠ 0` and  `a` is a square
   - `χ a = -1` otherwise

## Notation

Throughout most of this file, `F` denotes a finite field,
and `q` is notation for the cardinality of `F`, and `p` is the character of `F`.
`χ a` denotes the quadratic character of `a : F`.

-/

open_locale big_operators
open finset fintype monoid_hom nat.modeq function

namespace finite_field

variables {F : Type*} [field F] [fintype F] {p : ℕ} [char_p F p]
local notation `q` := fintype.card F -- declares `q` as a notation

/- ## basic -/
section basic

/-- `|F| = |units F| + 1`. The `+` version of `finite_field.card_units`. -/
lemma card_units' :
fintype.card F = fintype.card (units F) + 1 :=
begin
  simp [card_units],
  have h : 0 < fintype.card F := fintype.card_pos_iff.2 ⟨0⟩,
  rw [nat.sub_add_cancel h]
end

variables (p)
/-- If `n` is a prime number and `↑n = 0` in `F`, then `p = n`. -/
lemma char_eq_of_prime_eq_zero
{n : ℕ} (h₁ : nat.prime n) (h₂ : ↑n = (0 : F)) : p = n :=
begin
  have g₁ : nat.prime p, {obtain ⟨n, h1, h2⟩:= finite_field.card F p, assumption},
  have g₂ := (char_p.cast_eq_zero_iff F p n).1 h₂,
  exact (nat.prime_dvd_prime_iff_eq g₁ h₁).1 g₂,
end

lemma char_ne_two_of :
q ≡ 1 [MOD 4] ∨ q ≡ 3 [MOD 4] → p ≠ 2 :=
begin
  obtain ⟨n, p_prime, g⟩:= finite_field.card F p,
  rw g,
```

```
  rintro (h | h),
  any_goals
  { rw nat.modeq_iff_dvd at h,
    rintro hp, rw hp at h,
    rcases h with ⟨k, h⟩,
    simp at h,
    have : 2 ∣ 4 * k + 2 ^ ↑n,
    { use (2 * k + (2 : ℤ) ^ n.1.pred),
      simp [mul_add], congr' 1,
      ring,
      have : (2 : ℤ) * 2 ^ (n : ℕ).pred = 2 ^ ((n : ℕ).pred + 1), {refl},
      convert this.symm,
      clear this,
      convert (nat.succ_pred_eq_of_pos _).symm,
      exact n.2 },
  },
  { have g := eq_add_of_sub_eq h, rw ← g at this, revert this, dec_trivial },
  { have g := eq_add_of_sub_eq h, rw ← g at this, revert this, dec_trivial },
end

lemma char_ne_two_of' :
q ≡ 3 [MOD 4] → p ≠ 2 ∧ ¬ q ≡ 1 [MOD 4] :=
begin
  intros h,
  refine ⟨char_ne_two_of p (or.inr h), _⟩,
  intro h',
  have g := h'.symm.trans  h,
  simp [nat.modeq_iff_dvd] at g,
  rcases g with ⟨k, g⟩,
  norm_num at g,
  have : 2 / 4 = (4 * k) / 4, {rw ← g},
  norm_num at this,
  simp [← this] at g,
  assumption
end

/-- For non-zero `a : F`, `to_unit` converts `a` to an instance of `units F`. -/
@[simp] def to_unit {a : F} (h : a ≠ 0): units F :=
by refine {val := a, inv := a⁻¹, ..}; simp* at *

/-- The order of `a : units F` equals the order of `a` coerced in `F`. -/
lemma order_of_unit_eq (a : units F) : order_of a = order_of (a : F) :=
begin
  convert (@order_of_injective (units F) _ F _ (units.coe_hom F) _ a).symm,
  intros i j h,
  simp[*, units.ext_iff] at *,
end

variables {p}

/-- If the character of `F` is not `2`, `-1` is not equal to `1` in `F`. -/
lemma neg_one_ne_one_of (hp: p ≠ 2) : (-1 : F) ≠ 1 :=
begin
  intros h,
  have h' := calc ↑(2 : ℕ) = (2 : F)     : by simp
                      ... = (1 : F) + 1  : by ring
                      ... = (-1 : F) + 1 : by nth_rewrite 0 ←h
                      ... = 0            : by ring,
```

```
  have hp' := char_eq_of_prime_eq_zero p nat.prime_two h',
  contradiction
end

/-- If the character of `F` is not `2`, `-1` is not equal to `1` in `units F`. -/
lemma neg_one_ne_one_of' (hp: p ≠ 2) : (-1 : units F) ≠ 1 :=
by simp [units.ext_iff]; exact neg_one_ne_one_of hp

/-- For `x : F`, `x^2 = 1 ↔ x = 1 ∨ x = -1`. -/
lemma sq_eq_one_iff_eq_one_or_eq_neg_one (x : F) :
x^2 = 1 ↔ x = 1 ∨ x = -1 :=
begin
  rw [sub_eq_zero.symm],
  have h: x ^ 2 - 1 = (x - 1) * (x + 1), {ring},
  rw [h, mul_eq_zero, sub_eq_zero, add_eq_zero_iff_eq_neg]
end

/-- For `x : units F`, `x^2 = 1 ↔ x = 1 ∨ x = -1`. -/
lemma sq_eq_one_iff_eq_one_or_eq_neg_one' (x : units F) :
x^2 = 1 ↔ x = 1 ∨ x = -1 :=
begin
  simp only [units.ext_iff],
  convert sq_eq_one_iff_eq_one_or_eq_neg_one (x : F),
  simp,
end

/-- If the character of `F` is not `2`, `-1` is the only order 2 element in `F`. -/
lemma order_of_eq_two_iff (hp: p ≠ 2) (x : F) : order_of x = 2 ↔ x = -1 :=
begin
  have g := pow_order_of_eq_one x, -- g: x ^ order_of x = 1
  split, -- splits into `order_of x = 2 → x = -1` and `x = -1 → order_of x = 2`
  any_goals {intros h},
  { rw [h, sq_eq_one_iff_eq_one_or_eq_neg_one] at g, -- g : x = 1 ∨ x = -1
    have : x ≠ 1, { intro hx, rw ←order_of_eq_one_iff at hx, linarith },
    exact or_iff_not_imp_left.1 g this },
  { have hx : x^2 = 1, {rw h, ring},
    have inst : fact (nat.prime 2),
    {exact ⟨nat.prime_two⟩},
    resetI, -- resets the instance cache.
    convert order_of_eq_prime hx _,
    rw h,
    exact neg_one_ne_one_of hp }
end

/-- The "units" version of `order_of_eq_two_iff`. -/
lemma order_of_eq_two_iff' (hp: p ≠ 2) (x : units F) : order_of x = 2 ↔ x = -1 :=
by simp [units.ext_iff]; rw [← order_of_eq_two_iff hp (x : F), order_of_unit_eq x]

lemma card_eq_one_mod_n_iff_n_divide_card_units (n : ℕ):
q ≡ 1 [MOD n] ↔ n | fintype.card (units F) :=
begin
  rw [finite_field.card_units, nat.modeq.comm],
  convert nat.modeq_iff_dvd' (card_pos_iff.mpr ⟨(0 : F)⟩),
end

/-- `-1` is a square in `units F` iff the cardinal `q ≡ 1 [MOD 4]`. -/
theorem neg_one_eq_sq_iff' (hp: p ≠ 2) :
(∃ a : units F, -1 = a^2) ↔ q ≡ 1 [MOD 4] :=
```

```
begin
  -- rewrites the RHS to `4 | fintype.card (units F)`
  rw card_eq_one_mod_n_iff_n_divide_card_units,
  split, -- splits the goal into two directions
  -- the `→` direction
  { rintros ⟨a, h'⟩,
    -- h: a ^ 4 = 1
    have h := calc a^4 = a^2 * a^2 : by group
                ... = 1: by simp [← h'],
    -- h₀: order_of a | 4
    have h₀ := order_of_dvd_of_pow_eq_one h,
    -- au: order_of a ≤ 4
    have au := nat.le_of_dvd dec_trivial h₀,
    -- g₁ : a ≠ 1
    have g₁ : a ≠ 1,
    { rintro rfl, simp at h',
      exact absurd h' (neg_one_ne_one_of' hp) },
    -- h₁ : order_of a ≠ 1
    have h₁ := mt order_of_eq_one_iff.1 g₁,
    -- g₂ : a ≠ -1
    have g₂ : a ≠ -1,
    { rintro rfl, simp [pow_two] at h',
      exact absurd h' (neg_one_ne_one_of' hp) },
    -- h₂ : order_of a ≠ 2
    have h₂ := mt (order_of_eq_two_iff' hp a).1 g₂,
    -- ha : order_of a = 4
    have ha : order_of a = 4,
    { revert h₀ h₁ h₂,
      interval_cases (order_of a),
      any_goals {rw h_1, norm_num} },
    simp [← ha, order_of_dvd_card_univ] },
  -- the `←` direction
  { rintro ⟨k, hF⟩, -- hF : |units F| = 4 * k
    -- `hg'` is a proof that `g` generates `units F`
    obtain ⟨g, hg'⟩ := is_cyclic.exists_generator (units F),
    -- hg : g ^ |units F| = 1
    have hg := @pow_card_eq_one (units F) g _ _,
    have eq : 4 * k = k * 2 * 2, {ring},
    -- rewrite `hg` to `hg : g ^ (k * 2) = 1 ∨ g ^ (k * 2) = -1`
    rw [hF, eq, pow_mul, sq_eq_one_iff_eq_one_or_eq_neg_one'] at hg,
    rcases hg with (hg | hg), -- splits into two cases
    -- case `g ^ (k * 2) = 1`
    { have hk₁ := card_pos_iff.mpr ⟨(1 : units F)⟩,
      have o₁ := order_of_eq_card_of_forall_mem_gpowers hg',
      have o₂ := order_of_dvd_of_pow_eq_one hg,
      have le := nat.le_of_dvd (by linarith) o₂,
      rw [o₁, hF] at *,
      exfalso, linarith },
    -- case `g ^ (k * 2) = -1`
    { use g ^ k, simp [← hg, pow_mul] } },
end

/-- `-1` is a square in `F` iff the cardinal `q ≡ 1 [MOD 4]`. -/
lemma neg_one_eq_sq_iff (hp: p ≠ 2) :
(∃ a : F, -1 = a^2) ↔ fintype.card F ≡ 1 [MOD 4] :=
begin
  rw [←neg_one_eq_sq_iff' hp],
  -- the current goal is
```

```
  -- `(∃ (a : F), -1 = a ^ 2) ↔ ∃ (a : units F), -1 = a ^ 2`
  split, -- splits into two directions
  any_goals {rintros ⟨a, ha⟩},
  -- the `→` direction
  { have ha' : a ≠ 0,
    {rintros rfl, simp* at *},
    use (to_unit ha'),
    simpa [units.ext_iff] },
  -- the `←` direction
  { use a, simp [units.ext_iff] at ha, assumption },
  assumption
end

variables (F)

lemma disjoint_units_zero : disjoint {a : F | a ≠ 0} {0} :=
by simp

lemma units_union_zero : {a : F | a ≠ 0} ∪ {0} = @set.univ F :=
by {tidy, tauto}

end basic
/- ## end basic -/

/- ## quad_residues -/
section quad_residues

variables {F p}

/-- A proposition predicating whether a given `a : F` is a quadratic residue in `F`.
    `finite_field.is_quad_residue a` is defined to be `a ≠ 0 ∧ ∃ b, a = b^2`. -/
--@[derive decidable]
def is_quad_residue (a : F) : Prop := a ≠ 0 ∧ ∃ b, a = b^2

/-- A proposition predicating whether a given `a : F` is a quadratic non-residue in `F`.
    `finite_field.is_non_residue a` is defined to be `a ≠ 0 ∧ ¬ ∃ b, a = b^2`. -/
--@[derive decidable]
def is_non_residue (a : F) : Prop := a ≠ 0 ∧ ¬ ∃ b, a = b^2

instance [decidable_eq F] (a : F) : decidable (is_quad_residue a) :=
by {unfold is_quad_residue, apply_instance}

instance [decidable_eq F] (a : F) : decidable (is_non_residue a) :=
by {unfold is_non_residue, apply_instance}

variables {F p}

lemma not_residue_iff_is_non_residue
{a : F} (h : a ≠ 0) :
¬ is_quad_residue a ↔ is_non_residue a :=
by simp [is_quad_residue, is_non_residue, *] at *

lemma is_non_residue_of_not_residue
{a : F} (h : a ≠ 0) (g : ¬ is_quad_residue a) :
is_non_residue a :=
(not_residue_iff_is_non_residue h).1 g

lemma not_non_residue_iff_is_residue
```

```
{a : F} (h : a ≠ 0) :
¬ is_non_residue a ↔ is_quad_residue a :=
by simp [is_quad_residue, is_non_residue, *] at *

lemma is_residue_of_not_non_residue
{a : F} (h : a ≠ 0) (g : ¬ is_non_residue a) :
is_quad_residue a :=
(not_non_residue_iff_is_residue h).1 g

lemma residue_or_non_residue
{a : F} (h : a ≠ 0) :
is_quad_residue a ∨  is_non_residue a :=
begin
  by_cases g: is_quad_residue a,
  exact or.inl g,
  exact or.inr (is_non_residue_of_not_residue h g),
end

/-- `-1` is a residue if `q ≡ 1 [MOD 4]`. -/
lemma neg_one_is_residue_of (hF : q ≡ 1 [MOD 4]) :
is_quad_residue (-1 : F) :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- derive the char p of F
  resetI, -- resets the instance cache
  have hp := char_ne_two_of p (or.inl hF), -- hp: p ≠ 2
  have h := (neg_one_eq_sq_iff hp).2 hF, -- h : -1 is a square
  refine ⟨by tidy, h⟩
end

/-- `-1` is a non-residue if `q ≡ 3 [MOD 4]`. -/
lemma neg_one_is_non_residue_of (hF : q ≡ 3 [MOD 4]) :
is_non_residue (-1 : F) :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- derive the char p of F
  resetI, -- resets the instance cache
  -- hp: p ≠ 2, hF': ¬fintype.card F ≡ 1 [MOD 4]
  obtain ⟨hp, hF'⟩ := char_ne_two_of' p hF,
  -- h: ¬∃ (a : F), -1 = a ^ 2
  have h := mt (neg_one_eq_sq_iff hp).1 hF',
  refine ⟨by tidy, h⟩
end

variable (F)

@[simp] lemma eq_residues :
{j // ¬j = 0 ∧ ∃ (b : F), j = b ^ 2} =
{a : F // is_quad_residue a} := rfl

@[simp] lemma eq_non_residues :
{j // ¬j = 0 ∧ ¬∃ (x : F), j = x ^ 2} =
{a : F // is_non_residue a} := rfl

@[simp] lemma eq_non_residues' :
{j // ¬j = 0 ∧ ∀ (x : F), ¬j = x ^ 2} =
{a : F // is_non_residue a} := by simp [is_non_residue]

/-- `|F| = 1 + |{a : F // is_quad_residue a}| + |{a : F // is_non_residue a}|` -/
lemma eq_one_add_card_residues_add_card_non_residues
```

```
[decidable_eq F]:
q = 1 + fintype.card {a : F // is_quad_residue a} +
        fintype.card {a : F // is_non_residue a} :=
begin
  rw [fintype.card_split (λ a : F, a = 0),
      fintype.card_split' (λ a : F, a ≠ 0) (λ a, ∃ b, a = b^2)],
  simp [add_assoc],
end

/- ### sq_function -/
section sq_function

open quotient_group

variables (F) -- re-declares `F` as an explicit variable

/-- `sq` is the square function from `units F` to `units F`,
    defined as a group homomorphism. -/
def sq : (units F) →* (units F) :=
⟨λ a, a * a, by simp, (λ x y, by simp [units.ext_iff]; ring)⟩

/-- `|units F| = |(sq F).range| * |(sq F).ker|` -/
theorem sq.iso [decidable_eq F] :
fintype.card (units F) = fintype.card (sq F).range * fintype.card (sq F).ker :=
begin
  have iso := quotient_ker_equiv_range (sq F),
  have eq := fintype.card_congr (mul_equiv.to_equiv iso),
  rw [subgroup.card_eq_card_quotient_mul_card_subgroup (sq F).ker, eq]
end

/-- `sq.range_equiv` constructs the natural equivalence between
    the `(sq F).range` and `{a : F // is_quad_residue a}`. -/
def sq.range_equiv : (sq F).range ≃ {a : F // is_quad_residue a} :=
⟨
 λ a, ⟨a, by {have h := a.2, simp [sq] at h, rcases h with ⟨b, h⟩,
              simp [is_quad_residue, ← h], use b, ring }⟩,
 λ a, ⟨to_unit (a.2).1, by {obtain ⟨h, ⟨b, hab⟩⟩:= a.2,
                            have hb : b ≠ 0, {rintros rfl, simp* at *},
                            use to_unit hb, simp [units.ext_iff, sq, ← pow_two, ← hab] }⟩,
 λ a, by simp [units.ext_iff],
 λ a, by simp
⟩

/-- `|(sq F).range| = |{a : F // is_quad_residue a}|` -/
theorem sq.card_range_eq [decidable_eq F] :
fintype.card (sq F).range = fintype.card {a : F // is_quad_residue a} :=
by apply fintype.card_congr (sq.range_equiv F)

lemma sq.ker_carrier_eq :
(sq F).ker.carrier = {1, -1} :=
begin
  simp [ker, subgroup.comap, set.preimage, sq],
  ext, simp,
  convert sq_eq_one_iff_eq_one_or_eq_neg_one' x,
  simp [npow_rec],
end

lemma sq.card_ker_carrier_eq [decidable_eq F] (hp: p ≠ 2) :
```

```
fintype.card (sq F).ker.carrier = 2 :=
begin
  simp [sq.ker_carrier_eq F],
  convert @set.card_insert _ (1 : units F) {-1} _ _ _; simp,
  exact ne.symm (neg_one_ne_one_of' hp),
end

/-- `|(sq F).ker| = 2` if `p ≠ 2` -/
theorem sq.card_ker_eq [decidable_eq F] (hp: p ≠ 2) :
fintype.card (sq F).ker = 2 :=
by rw [←sq.card_ker_carrier_eq F hp]; refl

end sq_function
/- ### end sq_function -/

variable (F)

/-- `|units F| = |{a : F // is_quad_residue a}| * 2` -/
theorem card_units_eq_card_residues_mul_two [decidable_eq F] (hp: p ≠ 2) :
fintype.card (units F) = fintype.card {a : F // is_quad_residue a} * 2 :=
by rwa [sq.iso, sq.card_range_eq, sq.card_ker_eq F hp]

/-- `|{a : F // is_quad_residue a}| = |{a : F // is_non_residue a}|`-/
theorem card_residues_eq_card_non_residues
[decidable_eq F] (hp : p ≠ 2):
fintype.card {a : F // is_quad_residue a} =
fintype.card {a : F // is_non_residue a} :=
begin
  have eq := eq_one_add_card_residues_add_card_non_residues F,
  simp [card_units', card_units_eq_card_residues_mul_two F hp] at eq,
  linarith
end
/- `card_units_eq_card_residues_mul_two F hp` is a built API that proves
   `|F| = 1 + |{a : F // is_quad_residue a}| + |{a : F // is_non_residue a}|` -/

/-- unfolded version of `card_residues_eq_card_non_residues` -/
theorem card_residues_eq_card_non_residues'
[decidable_eq F] (hp : p ≠ 2):
(@univ {a : F // is_quad_residue a} _).card =
(@univ {a : F // is_non_residue a} _).card :=
by convert card_residues_eq_card_non_residues F hp


variable {F} -- re-declares `F` as an implicit variable

example : (0 : F)⁻¹ = 0 := by simp

/-- `a⁻¹` is a residue if and only if `a` is. -/
theorem inv_is_residue_iff {a : F} :
is_quad_residue a⁻¹ ↔ is_quad_residue a :=
begin
  split, -- splits into two directions
  any_goals {rintro ⟨h, b, g⟩, refine ⟨by tidy, b⁻¹, by simp [←g]⟩},
end

/-- `a⁻¹` is a non residue if and only if `a` is. -/
theorem inv_is_non_residue_iff {a : F} :
is_non_residue a⁻¹ ↔ is_non_residue a :=
```

```
begin
  by_cases h : a = 0,
  {simp* at *}, -- when `a = 0`
  have h' : a⁻¹ ≠ 0 := by simp [h],
  simp [←not_residue_iff_is_non_residue, *, inv_is_residue_iff]
end

theorem residue_mul_residue_is_residue
{a b : F} (ha : is_quad_residue a) (hb : is_quad_residue b) :
is_quad_residue (a * b) :=
begin
  obtain ⟨ha, c, rfl⟩ := ha,
  obtain ⟨hb, d, rfl⟩ := hb,
  refine ⟨mul_ne_zero ha hb, c*d, _⟩,
  ring
end

theorem non_residue_mul_residue_is_non_residue
{a b : F} (ha : is_non_residue a) (hb : is_quad_residue b) :
is_non_residue (a * b) :=
begin
  obtain ⟨hb, c, rfl⟩ := hb,
  refine ⟨mul_ne_zero ha.1 hb, _⟩,
  rintro ⟨d, h⟩,
  convert ha.2 ⟨(d * c⁻¹), _⟩,
  field_simp [← h],
end

theorem residue_mul_non_residue_is_non_residue
{a b : F} (ha : is_quad_residue a) (hb : is_non_residue b):
is_non_residue (a * b) :=
by simp [mul_comm a, non_residue_mul_residue_is_non_residue hb ha]

/-- `finite_filed.non_residue_mul` is the map `a * _` given a non-residue `a`
    defined on `{b : F // is_quad_residue b}`. -/
def non_residue_mul {a: F} (ha : is_non_residue a) :
{b : F // is_quad_residue b} → {b : F // is_non_residue b} :=
λ b, ⟨a * b, non_residue_mul_residue_is_non_residue ha b.2⟩

open function

/-- proves that `a * _` is injective from residues for a non-residue `a` -/
lemma is_non_residue.mul_is_injective
{a: F} (ha : is_non_residue a) :
injective (non_residue_mul ha):=
begin
  intros b₁ b₂ h,
  simp [non_residue_mul] at h,
  ext,
  convert or.resolve_right h ha.1,
end

/-- proves that `a * _` is surjective onto non-residues for a non-residue `a` -/
lemma is_non_residue.mul_is_surjective
[decidable_eq F] (hp : p ≠ 2) {a: F} (ha : is_non_residue a) :
surjective (non_residue_mul ha):=
begin
  by_contra, -- prove by contradtiction
```

```
  have lt := card_lt_of_injective_not_surjective
    (non_residue_mul ha) (ha.mul_is_injective) h,
  have eq := card_residues_eq_card_non_residues F hp,
  linarith
end

theorem non_residue_mul_non_residue_is_residue
[decidable_eq F] (hp : p ≠ 2)
{a b : F} (ha : is_non_residue a) (hb : is_non_residue b):
is_quad_residue (a * b) :=
begin
  by_contra h, -- prove by contradtiction
  -- rw `h` to `is_non_residue (a * b)`
  rw [not_residue_iff_is_non_residue (mul_ne_zero ha.1 hb.1)] at h,
  -- surj : `a * _` is surjective onto non-residues from residues
  have surj := ha.mul_is_surjective hp,
  simp [function.surjective] at surj,
  -- in particular, non-residue `a * b` is in the range of `a * _`
  specialize surj (a * b) h,
  -- say `a * b' = a * b` and `hb' : is_quad_residue b'`
  rcases surj with ⟨b', hb', eq⟩,
  simp [non_residue_mul, ha.1] at eq, -- `eq: b' = b`
  rw [eq] at hb',
  exact absurd hb'.2 hb.2,
end

end quad_residues

/- ## end quad_residues -/

/- ## quad_char -/

section quad_char

variables {F} [decidable_eq F]

/-- `finite_field.quad_char` defines the quadratic character of `a` in a finite field `F`. -/
def quad_char (a : F) : ℚ :=
if a = 0            then 0 else
if ∃ b : F, a = b^2 then 1 else
                         -1.

notation `χ` := quad_char  -- declare the notation for `quad_char`

@[simp] lemma quad_char_zero_eq_zero : χ (0 : F) = 0 := by simp [quad_char]

@[simp] lemma quad_char_one_eq_one : χ (1 : F) = 1 :=
by {simp [quad_char], intros h, specialize h 1, simp* at *}

@[simp] lemma quad_char_ne_zero_of_ne_zero {a : F} (h : a ≠ 0) :
χ a ≠ 0 :=
by by_cases (∃ (b : F), a = b ^ 2); simp [quad_char, *] at *

@[simp] lemma quad_char_eq_zero_iff_eq_zero (a : F) :
χ a = 0 ↔ a = 0 :=
begin
  split,
  {contrapose, exact quad_char_ne_zero_of_ne_zero},
```

```
  {rintro rfl, exact quad_char_zero_eq_zero}
end

@[simp] lemma quad_char_i_sub_j_eq_zero_iff_i_eq_j (i j : F) :
χ (i - j) = 0 ↔ i = j :=
by simp [sub_eq_zero]

@[simp] lemma quad_char_eq_one_or_neg_one_of_char_ne_zero {a : F} (h : χ a ≠ 0) :
(χ a = 1) ∨ (χ a = -1) :=
by by_cases (∃ (b : F), a = b ^ 2); simp [quad_char, *] at *

@[simp] lemma quad_char_eq_neg_one_or_one_of_ne_zero' {a : F} (h : χ a ≠ 0) :
(χ a = -1) ∨ (χ a = 1) :=
or.swap (quad_char_eq_one_or_neg_one_of_char_ne_zero h)

@[simp] lemma quad_char_eq_one_or_neg_one_of_ne_zero {a : F} (h : a ≠ 0) :
(χ a = 1) ∨ (χ a = -1) :=
quad_char_eq_one_or_neg_one_of_char_ne_zero (quad_char_ne_zero_of_ne_zero h)

@[simp] lemma quad_char_suqare_eq_one_of_ne_zero {a : F} (h : a ≠ 0) :
χ a * χ a = 1 :=
by by_cases (∃ (b : F), a = b ^ 2); simp [quad_char, *] at *

@[simp] lemma quad_char_eq_one_of {a : F} (h : a ≠ 0) (h' : ∃ (b : F), a = b ^ 2) :
χ a = 1 :=
by simp [quad_char, *] at *

@[simp] lemma quad_char_eq_neg_one_of {a : F} (h : a ≠ 0) (h' : ¬ ∃ (b : F), a = b ^ 2) :
χ a = -1 :=
by simp [quad_char, *] at *

@[simp] lemma quad_char_eq_one_of_quad_residue {a : F} (h : is_quad_residue a) :
χ a = 1 := quad_char_eq_one_of h.1 h.2

@[simp] lemma quad_char_eq_neg_one_of_non_residue {a : F} (h : is_non_residue a) :
χ a = -1 := quad_char_eq_neg_one_of h.1 h.2

@[simp] lemma nonzero_quad_char_eq_one_or_neg_one' {a b : F} (h : a ≠ b) :
(χ (a - b) = 1) ∨ (χ (a - b) = -1) :=
by {have h'':= sub_ne_zero.mpr h, simp* at *}

variables {F p}

theorem quad_char_inv (a : F) : χ a⁻¹ = χ a :=
begin
  by_cases ha: a = 0,
  {simp [ha]}, -- case `a=0`
  -- splits into cases if `a` is a residue
  obtain (ga | ga) := residue_or_non_residue ha,
  -- case 1: `a` is a residue
  {have g':= inv_is_residue_iff.2 ga, simp*},
  -- case 2: `a` is a non-residue
  {have g':= inv_is_non_residue_iff.2 ga, simp*},
end

theorem quad_char_mul (hp : p ≠ 2) (a b : F) : χ (a * b) = χ a * χ b :=
begin
  by_cases ha: a = 0,
```

```
  any_goals {by_cases hb : b = 0},
  any_goals {simp*}, -- closes cases when `a = 0` or `b =0`
  -- splits into cases if `a` is a residue
  obtain (ga | ga) := residue_or_non_residue ha,
  -- splits into cases if `b` is a residue
  any_goals {obtain (gb | gb) := residue_or_non_residue hb},
  -- case 1 : `a` is, `b` is
  { have g:= residue_mul_residue_is_residue ga gb, simp* },
  -- case 2 : `a` is, `b` is not
  { have g:= residue_mul_non_residue_is_non_residue ga gb, simp* },
  -- case 1 : `a` is not, `b` is
  { have g:= non_residue_mul_residue_is_non_residue ga gb, simp* },
  -- case 4 : `a` is not, `b` is not
  { have g:= non_residue_mul_non_residue_is_residue hp ga gb, simp* },
end

/-- `χ (-1) = 1` if `q ≡ 1 [MOD 4]`. -/
@[simp] theorem char_neg_one_eq_one_of (hF : q ≡ 1 [MOD 4]) :
χ (-1 : F) = 1 :=
by simp [neg_one_is_residue_of hF]

/-- `χ (-1) = -1` if `q ≡ 3 [MOD 4]`. -/
@[simp] theorem char_neg_one_eq_neg_one_of (hF : q ≡ 3 [MOD 4]) :
χ (-1 : F) = -1 :=
by simp [neg_one_is_non_residue_of hF]

/-- `χ (-i) = χ i` if `q ≡ 1 [MOD 4]`. -/
theorem quad_char_is_sym_of (hF : q ≡ 1 [MOD 4]) (i : F) :
χ (-i) = χ i :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- derive the char p of F
  resetI, -- resets the instance cache
  have hp := char_ne_two_of p (or.inl hF), -- hp: p ≠ 2
  have h := char_neg_one_eq_one_of hF, -- h: χ (-1) = 1
  -- χ (-i) = 1 * χ (-i) = χ (-1) * χ (-i) = χ ((-1) * (-i))
  rw [← one_mul (χ (-i)), ← h, ← quad_char_mul hp],
  simp, assumption
end

/-- another form of `quad_char_is_sym_of` -/
theorem quad_char_is_sym_of' (hF : q ≡ 1 [MOD 4]) (i j : F) :
χ (j - i) = χ (i - j) :=
by convert quad_char_is_sym_of hF (i - j); ring

/-- `χ (-i) = - χ i` if `q ≡ 3 [MOD 4]`. -/
theorem quad_char_is_skewsym_of (hF : q ≡ 3 [MOD 4]) (i : F) :
χ (-i) = - χ i :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- derive the char p of F
  resetI, -- resets the instance cache
  have hp := char_ne_two_of p (or.inr hF), -- hp: p ≠ 2
  have h := char_neg_one_eq_neg_one_of hF, -- h: χ (-1) = 1
  rw [← neg_one_mul (χ i), ← h, ← quad_char_mul hp],
  simp, assumption
end

/-- another form of `quad_char_is_skewsym_of` -/
theorem quad_char_is_skewsym_of' (hF : q ≡ 3 [MOD 4]) (i j : F) :
```

```
χ (j - i) = - χ (i - j) :=
by convert quad_char_is_skewsym_of hF (i - j); ring

variable (F) -- use `F` as an explicit parameter

/-- `∑ a : {a : F // a ≠ 0}, χ (a : F) = 0` if `p ≠ 2`. -/
lemma quad_char.sum_in_non_zeros_eq_zero (hp : p ≠ 2):
∑ a : {a : F // a ≠ 0}, χ (a : F) = 0 :=
begin
  simp [fintype.sum_split' (λ a : F, a ≠ 0) (λ a : F, ∃ b, a = b^2)],
  suffices h :
  ∑ (j : {j // j ≠ 0 ∧ ∃ (b : F), j = b ^ 2}), χ (j : F) =
  ∑ a : {a : F // is_quad_residue a} , 1,
  suffices g :
  ∑ (j : {j // j ≠ 0 ∧ ¬∃ (b : F), j = b ^ 2}), χ (j : F) =
  ∑ a : {a : F // is_non_residue a} , -1,
  simp [h, g, sum_neg_distrib,
        card_residues_eq_card_non_residues' F hp],
  any_goals
  {apply fintype.sum_congr, intros a, have := a.2, simp* at *},
end

/-- `∑ (a : F), χ a = 0` if `p ≠ 2`. -/
theorem quad_char.sum_eq_zero (hp : p ≠ 2):
∑ (a : F), χ a = 0 :=
by simp [fintype.sum_split (λ b, b = (0 : F)),
         quad_char.sum_in_non_zeros_eq_zero F hp, default]

variable {F} -- use `F` as an implicit parameter

/-- another form of `quad_char.sum_eq_zero` -/
@[simp] lemma quad_char.sum_eq_zero_reindex_1 (hp : p ≠ 2) {a : F}:
∑ (b : F), χ (a - b) = 0 :=
begin
  rw ← quad_char.sum_eq_zero F hp,
  refine fintype.sum_equiv ((equiv.sub_right a).trans (equiv.neg _)) _ _ (by simp),
end

/-- another form of `quad_char.sum_eq_zero` -/
@[simp] lemma quad_char.sum_eq_zero_reindex_2 (hp : p ≠ 2) {b : F}:
∑ (a : F), χ (a - b) = 0 :=
begin
  rw ← quad_char.sum_eq_zero F hp,
  refine fintype.sum_equiv (equiv.sub_right b) _ _ (by simp),
end

variable {F} -- use `F` as an implicit parameter

/-- helper of `quad_char.sum_mul'`: reindex the terms in the summation -/
lemma quad_char.sum_mul'_aux {b : F} (hb : b ≠ 0) :
∑ (a : F) in filter (λ (a : F), ¬a = 0) univ, χ (1 + a⁻¹ * b) =
∑ (c : F) in filter (λ (c : F), ¬c = 1) univ, χ (c) :=
begin
  refine finset.sum_bij (λ a ha, 1 + a⁻¹ * b) (λ a ha, _)
    (λ a ha, rfl) (λ a₁ a₂ h1 h2 h, _) (λ c hc, _),
  { simp at ha, simp* },
  { simp at h1 h2, field_simp at h, rw (mul_right_inj' hb).1 h.symm },
  { simp at hc, use b * (c - 1)⁻¹,
```

```
    simp [*, mul_inv_rev', sub_ne_zero.2 hc] }
end

/-- If `b ≠ 0` and `p ≠ 2`, `∑ a : F, χ (a) * χ (a + b) = -1`. -/
theorem quad_char.sum_mul' {b : F} (hb : b ≠ 0) (hp : p ≠ 2):
∑ a : F, χ (a) * χ (a + b) = -1 :=
begin
  rw [finset.sum_split _ (λ a, a = (0 : F))],
  simp,
  have h : ∑ (a : F) in filter (λ (a : F), ¬a = 0) univ, χ a * χ (a + b) =
           ∑ (a : F) in filter (λ (a : F), ¬a = 0) univ, χ (1 + a⁻¹ * b),
  { apply finset.sum_congr rfl,
    intros a ha, simp at ha,
    simp [←quad_char_inv a, ←quad_char_mul hp a⁻¹],
    field_simp },
  rw [h, quad_char.sum_mul'_aux hb],
  have g:= @finset.sum_split _ _ _ (@finset.univ F _) (χ) (λ a : F, a = 1) _,
  simp [quad_char.sum_eq_zero F hp] at g,
  linarith
end

/-- another form of `quad_char.sum_mul'` -/
theorem quad_char.sum_mul {b c : F} (hbc : b ≠ c) (hp : p ≠ 2):
∑ a : F, χ (b - a) * χ (c - a) = -1 :=
begin
  rw ← quad_char.sum_mul' (sub_ne_zero.2 (ne.symm hbc)) hp,
  refine fintype.sum_equiv ((equiv.sub_right b).trans (equiv.neg _)) _ _ (by simp),
end

end quad_char
/- ## end quad_char -/

end finite_field
```

MAIN1.LEAN

```
/-
Copyright (c) 2021 ***. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: ***.
-/
import data.matrix.notation
import symmetric_matrix

/-!
# Hadamard product and Kronecker product.

This file defines the Hadamard product `matrix.Hadamard` and the Kronecker product `matrix.Kronecker` and
contains basic properties about them.

## Main definitions

- `matrix.Hadamard`: defines the Hadamard product, which is the pointwise product of two matrices
   of the same size.
- `matrix.Kronecker`: defines the Kronecker product, denoted by `⊗`.
  For `A : matrix I J α` and `B : matrix K L α`, `A ⊗ B ⟨a, b⟩ ⟨c, d⟩` is defined to be ` (A a c) * (B b d)`.
- `matrix.fin_Kronecker`: the `fin` version of `matrix.Kronecker`, denoted by `⊗□`.
  For `A : matrix (fin m) (fin n) α` and `B : matrix (fin p) (fin q) α`, `A ⊗□ B` is of type
```

```
  `matrix (fin (m * p)) (fin (n * q))`.
  The difference from `A ⊗ B` is that each of the index types `fin (m * p)` and `fin (n * q)`
  of the resulting matrix has a natural order.

## Notation

* `⊙`: the Hadamard product `matrix.Hadamard`;
* `⊗`: the Kronecker product `matrix.Kronecker`;
* `⊗□`: the Kronecker product `fin_Kronecker` of matrices with index types `fin _`.

## References

*  <https://en.wikipedia.org/wiki/Hadamard_product_(matrices)>
*  <https://en.wikipedia.org/wiki/Kronecker_product>

## Tags

Hadamard product, Kronecker product, Hadamard, Kronecker
-/

variables {α β γ I J K L M N: Type*}
variables {R : Type*}
variables {m n p q r s t: ℕ}
variables [fintype I] [fintype J] [fintype K] [fintype L] [fintype M] [fintype N]

namespace matrix
open_locale matrix big_operators
open complex

/- ## Hadamard product  -/
section Hadamard_product

def Hadamard [has_mul α] (A : matrix I J α) (B : matrix I J α) :
matrix I J α :=
λ i j, (A i j) * (B i j)
localized "infix `⊙`:100 := matrix.Hadamard" in matrix -- declares the notation

section basic_properties
variables (A : matrix I J α) (B : matrix I J α) (C : matrix I J α)

/- commutativity -/
lemma Had_comm [comm_semigroup α] : A ⊙ B = B ⊙ A :=
by ext; simp [Hadamard, mul_comm]

/- associativity -/
lemma Had_assoc [semigroup α] : A ⊙ B ⊙ C = A ⊙ (B ⊙ C) :=
by ext; simp [Hadamard, mul_assoc]

/- distributivity -/
section distrib
variables [distrib α]
lemma Had_add : A ⊙ (B + C) = A ⊙ B + A ⊙ C :=
by ext; simp [Hadamard, left_distrib]
lemma add_Had : (B + C) ⊙ A = B ⊙ A + C ⊙ A :=
by ext; simp [Hadamard, right_distrib]
end distrib

/- scalar multiplication -/
```

```
section scalar
@[simp] lemma smul_Had
[has_mul α] [has_scalar R α] [is_scalar_tower R α α] (k : R) :
(k • A) ⊙ B = k • A ⊙ B :=
by {ext, simp [Hadamard], exact smul_assoc _ (A i j) _}
@[simp] lemma Had_smul
[has_mul α] [has_scalar R α] [smul_comm_class R α α] (k : R):
A ⊙ (k • B) = k • A ⊙ B :=
by {ext, simp [Hadamard], exact (smul_comm k (A i j) (B i j)).symm}
end scalar

section zero
variables [mul_zero_class α]
@[simp] lemma Had_zero : A ⊙ (0 : matrix I J α) = 0 :=
by ext; simp [Hadamard]
@[simp] lemma zero_Had : (0 : matrix I J α) ⊙ A = 0 :=
by ext; simp [Hadamard]
end zero

section trace
variables [comm_semiring α] [decidable_eq I] [decidable_eq J]

/--`vᵀ (M₁ ⊙ M₂) w = tr ((diagonal v)ᵀ · M₁ · (diagonal w) · M₂ᵀ)` -/
lemma tr_identity (v : I → α) (w : J → α) (M₁ : matrix I J α) (M₂ : matrix I J α):
dot_product (vec_mul  v  (M₁ ⊙ M₂)) w =
tr ((diagonal v)ᵀ · M₁ · (diagonal w) · M₂ᵀ) :=
begin
  simp [dot_product, vec_mul, Hadamard, finset.sum_mul],
  rw finset.sum_comm,
  apply finset.sum_congr, refl, intros i hi,
  simp [diagonal, transpose, matrix.mul, dot_product],
  apply finset.sum_congr, refl, intros j hj,
  ring,
end

/-- `trace` version of `tr_identity` -/
lemma trace_identity (v : I → α) (w : J → α) (M₁ : matrix I J α) (M₂ : matrix I J α):
dot_product (vec_mul  v  (M₁ ⊙ M₂)) w =
trace I α α ((diagonal v)ᵀ · M₁ · (diagonal w) · M₂ᵀ) :=
by rw [trace_eq_tr, tr_identity]

/-- `∑ (i : I) (j : J), (M₁ ⊙ M₂) i j = tr (M₁ · M₂ᵀ)` -/
lemma sum_Had_eq_tr_mul (M₁ : matrix I J α) (M₂ : matrix I J α) :
∑ (i : I) (j : J), (M₁ ⊙ M₂) i j = tr (M₁ · M₂ᵀ) :=
begin
  have h:= tr_identity (λ i, 1 : I → α) (λ i, 1 : J → α) M₁ M₂,
  simp at h,
  rw finset.sum_comm at h,
  assumption,
end

/-- `vᴴ (M₁ ⊙ M₂) w = tr ((diagonal v)ᴴ · M₁ · (diagonal w) · M₂ᵀ)` over `ℂ` -/
lemma tr_identity_over_ℂ
(v : I → ℂ) (w : J → ℂ) (M₁ : matrix I J ℂ) (M₂ : matrix I J ℂ):
dot_product (vec_mul (star v)  (M₁ ⊙ M₂)) w =
tr ((diagonal v)ᴴ · M₁ · (diagonal w) · M₂ᵀ) :=
begin
  simp [dot_product, vec_mul, Hadamard, finset.sum_mul],
```

```
  rw finset.sum_comm,
  apply finset.sum_congr, refl, intros i hi,
  simp [diagonal, transpose, conj_transpose, matrix.mul, dot_product, star, has_star.star],
  apply finset.sum_congr, refl, intros j hj,
  ring_nf,
end

/-- `trace` version of `tr_identity_over_ℂ` -/
lemma trace_identity_over_ℂ
(v : I → ℂ) (w : J → ℂ) (M₁ : matrix I J ℂ) (M₂ : matrix I J ℂ):
dot_product (vec_mul (star v)  (M₁ ⊙ M₂)) w =
trace I ℂ ℂ ((diagonal v)ᴴ · M₁ · (diagonal w) · M₂ᵀ) :=
by rw [trace_eq_tr, tr_identity_over_ℂ]

end trace

end basic_properties

end Hadamard_product
/- ## end Hadamard product  -/

/- ## Kronecker product  -/
section Kronecker_product
open_locale matrix

@[elab_as_eliminator]
def Kronecker [has_mul α] (A : matrix I J α) (B : matrix K L α) :
matrix (I × K) (J × L) α :=
λ ⟨i, k⟩ ⟨j, l⟩, (A i j) * (B k l)

/- an infix notation for the Kronecker product -/
localized "infix `⊗`:100 := matrix.Kronecker" in matrix

/- The advantage of the following def is that one can directly #eval the Kronecker
   product of specific matrices-/
/- ## fin_Kronecker_prodcut  -/

@[elab_as_eliminator]
def fin_Kronecker [has_mul α]
(A : matrix (fin m) (fin n) α) (B : matrix (fin p) (fin q) α)
: matrix (fin (m * p)) (fin (n * q)) α :=
λ i j,
A ⟨(i / p), by {have h:= i.2, simp [mul_comm m] at *, apply nat.div_lt_of_lt_mul h}⟩
  ⟨(j / q), by {have h:= j.2, simp [mul_comm n] at *, apply nat.div_lt_of_lt_mul h}⟩
*
B ⟨(i % p), by {cases p, linarith [i.2], apply nat.mod_lt _ (nat.succ_pos _)}⟩
  ⟨(j % q), by {cases q, linarith [j.2], apply nat.mod_lt _ (nat.succ_pos _)}⟩

localized "infix `⊗□`:100 := matrix.fin_Kronecker" in matrix

section notations
def matrix_empty : matrix (fin 0) (fin 0) α := λ x, ![]
localized "notation `!![]` := matrix.matrix_empty" in matrix
example : (!![]: matrix (fin 0) (fin 0) α) = ![] := by {ext, have h:= x.2, simp* at *}
end notations

section examples
open_locale matrix
```

```
def ex1:= ![![1, 2], ![3, 4]]
def ex2:= ![![0, 5], ![6, 7]]
def ex3:= ![![(1:ℤ), -4, 7], ![-2, 3, 3]]
def ex4:= ![![(8:ℤ), -9, -6, 5], ![1, -3, -4, 7], ![2, 8, -8, -3], ![1, 2, -5, -1]]

#eval (!![]: matrix (fin 0) (fin 0) ℕ)
#eval ex3 ⊗ₖ ex4
#eval ex1 ⊗ₖ ex2
#eval 2 • (ex1 ⊗ₖ ex2)
#eval ex2 ⊗ₖ ![![]]
#eval ![![]] ⊗ₖ ex2
#eval ex2 ⊗ₖ !![]
#eval !![] ⊗ₖ ex2
#eval ![![]] ⊗ₖ (![![]] :matrix (fin 1) (fin 0) ℕ)

end examples
/- ## end fin_Kronecker_prodcut  -/

lemma Kronecker_apply [has_mul α]
(A : matrix I J α) (B : matrix K L α) (a : I × K) (b : J × L) :
(A ⊗ B) a b = (A a.1 b.1) * (B a.2 b.2) :=
begin
  have ha : a = ⟨a.1, a.2⟩ := by {ext; simp},
  have hb : b = ⟨b.1, b.2⟩ := by {ext; simp},
  rw [ha, hb], dsimp [Kronecker], refl
end

/- distributivity -/
section distrib
variables [distrib α] -- variables are restricted to this section
variables (A : matrix I J α) (B : matrix K L α) (B' : matrix K L α)
lemma K_add :A ⊗ (B + B') = A ⊗ B + A ⊗ B' :=
by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker, left_distrib]}
lemma add_K :(B + B') ⊗ A = B ⊗ A + B' ⊗ A :=
by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker, right_distrib]}
end distrib

/- distributivity over substraction -/
section distrib_sub
variables [ring α]
variables (A : matrix I J α) (B : matrix K L α) (B' : matrix K L α)
lemma K_sub :A ⊗ (B - B') = A ⊗ B - A ⊗ B' :=
by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker, mul_sub]}
lemma sub_K :(B - B') ⊗ A = B ⊗ A - B' ⊗ A :=
by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker, sub_mul]}
end distrib_sub

/-- associativity -/
lemma K_assoc
[semigroup α] (A : matrix I J α) (B : matrix K L α) (C : matrix M N α) :
A ⊗ B ⊗ C = A ⊗ (B ⊗ C) :=
by {ext ⟨⟨a1, b1⟩, c1⟩ ⟨⟨a2, b2⟩, c2⟩, simp[Kronecker, mul_assoc], refl}

section zero
variables [mul_zero_class α] (A : matrix I J α)
@[simp] lemma K_zero : A ⊗ (0 : matrix K L α) = 0 :=
by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker]}
@[simp] lemma K_zero' : A ⊗ ((λ _ _, 0):matrix K L α) = 0 :=
```

```
K_zero A
@[simp] lemma zero_K : (0 : matrix K L α) ⊗ A = 0 :=
by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker]}
@[simp] lemma zero_K' : ((λ _ _, 0):matrix K L α) ⊗ A = 0 :=
zero_K A
end zero

/-- `1 ⊗ 1 = 1`.
    The Kronecker product of two identity matrices is an identity matrix. -/
@[simp] lemma one_K_one
[mul_zero_one_class α] [decidable_eq I] [decidable_eq J] :
(1 :matrix I I α) ⊗ (1 :matrix J J α) = 1 :=
begin
  ext ⟨a,b⟩ ⟨c,d⟩,
  by_cases h: a = c,
  by_cases g: b = d},
  any_goals {simp[*, Kronecker] at *},
end

section neg
variables [ring α]
variables (A : matrix I J α) (B : matrix K L α)
@[simp] lemma neg_K: (-A) ⊗ B = - A ⊗ B := by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker]}
@[simp] lemma K_neg: A ⊗ (-B) = - A ⊗ B := by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker]}
end neg

/- scalar multiplication -/
section scalar
@[simp] lemma smul_K
[has_mul α] [has_scalar R α] [is_scalar_tower R α α]
(k : R) (A : matrix I J α) (B : matrix K L α) :
(k • A) ⊗ B = k • A ⊗ B :=
by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker], exact smul_assoc _ (A a c) _}
@[simp] lemma K_smul
[has_mul α] [has_scalar R α] [smul_comm_class R α α]
(k : R) (A : matrix I J α) (B : matrix K L α) :
A ⊗ (k • B) = k • A ⊗ B :=
by {ext ⟨a,b⟩ ⟨c,d⟩, simp [Kronecker], exact (smul_comm k (A a c) _).symm}
end scalar

/- Kronecker product mixes matrix multiplication -/
section Kronecker_mul
variables [comm_ring α]
variables
(A : matrix I J α) (C : matrix J K α)
(B : matrix L M α) (D : matrix M N α)
lemma K_mul: (A ⊗ B) ⬝ (C ⊗ D) = (A ⬝ C) ⊗ (B ⬝ D) :=
begin
  ext ⟨a,b⟩ ⟨c,d⟩,
  simp [matrix.mul,dot_product,Kronecker,finset.sum_mul,finset.mul_sum],
  rw [←finset.univ_product_univ,finset.sum_product],
  simp [Kronecker._match_1,Kronecker._match_2],
  rw finset.sum_comm,
  repeat {congr, ext},
  ring,
end
variables [decidable_eq I] [decidable_eq M] [decidable_eq L] [decidable_eq J]
@[simp] lemma one_K_mul: (1 ⊗ B) ⬝ (A ⊗ 1) = A ⊗ B := by simp [K_mul]
```

```
@[simp] lemma K_one_mul: (A ⊗ 1) · (1 ⊗ B) = A ⊗ B := by simp [K_mul]
end Kronecker_mul

/- Kronecker product mixes Hadamard product -/
section Kronecker_Hadamard
variables [comm_semigroup α]
(A : matrix I J α) (C : matrix I J α)
(B : matrix K L α) (D : matrix K L α)
lemma Kronecker_Hadamard : (A ⊗ B) ⊙ (C ⊗ D) = (A ⊙ C) ⊗ (B ⊙ D) :=
begin
  ext ⟨a, b⟩ ⟨c, d⟩,
  simp [Hadamard, Kronecker],
  rw ← mul_assoc,
  rw mul_assoc _ (B b d),
  rw mul_comm (B b d),
  simp [mul_assoc]
end
end Kronecker_Hadamard

lemma transpose_K
[has_mul α] (A : matrix I J α) (B : matrix K L α):
(A ⊗ B)ᵀ = Aᵀ ⊗ Bᵀ :=
by ext ⟨a,b⟩ ⟨c,d⟩; simp [transpose, Kronecker]

lemma conj_transpose_K
[comm_monoid α] [star_monoid α] (M₁ : matrix I J α) (M₂ : matrix K L α) :
(M₁ ⊗ M₂)ᴴ = M₁ᴴ ⊗ M₂ᴴ:=
by ext ⟨a,b⟩ ⟨c,d⟩; simp [conj_transpose,Kronecker, mul_comm]

section trace
variables [semiring β] [non_unital_non_assoc_semiring α] [module β α]
variables (A : matrix I I α) (B : matrix J J α)
lemma trace_K: trace (I × J) β α (A ⊗ B) = (trace I β α A) * (trace J β α B) :=
begin
  simp[Kronecker, trace, ←finset.univ_product_univ, finset.sum_product,
       Kronecker._match_2,finset.sum_mul,finset.mul_sum],
  rw finset.sum_comm,
end
end trace

section inverse

variables [decidable_eq I] [decidable_eq J] [comm_ring α]
variables (A : matrix I I α) (B : matrix J J α) (C : matrix I I α)

lemma K_inverse [invertible A] [invertible B] :(A ⊗ B)⁻¹ = A⁻¹ ⊗ B⁻¹ :=
begin
  suffices : (A⁻¹ ⊗ B⁻¹) · (A ⊗ B) = 1,
  apply inv_eq_left_inv this,
  simp [K_mul],
end

@[simp] noncomputable
def Kronecker.invertible_of_invertible [invertible A] [invertible B] :
invertible (A ⊗ B) :=
⟨A⁻¹ ⊗ B⁻¹, by simp [K_mul], by simp [K_mul]⟩

@[simp] lemma Kronecker.unit_of_unit (ha : is_unit A) (hb : is_unit B) :
```

```
is_unit (A ⊗ B) :=
@is_unit_of_invertible _ _
  (A ⊗ B)
  (@Kronecker.invertible_of_invertible _ _ _ _ _ _ _ _ A B
    (is_unit.invertible ha) (is_unit.invertible hb))

end inverse

section symmetric
variables [has_mul α]
@[simp] lemma Kronecker.is_sym_of_is_sym {A : matrix I I α} {B : matrix J J α}
  (ha: A.is_sym) (hb: B.is_sym) : (A ⊗ B).is_sym :=
  by simp [matrix.is_sym, transpose_K, *] at *
@[simp] lemma Kronecker.is_Hermitian_of_is_Hermitian {A : matrix I I ℂ} {B : matrix J J ℂ}
  (ha: A.is_Hermitian) (hb: B.is_Hermitian) : (A ⊗ B).is_Hermitian :=
  by simp [matrix.is_Hermitian, conj_transpose_K, *] at *
end symmetric

section ortho
variables  [decidable_eq I] [decidable_eq J]
@[simp] lemma Kronecker.is_ortho_of_is_ortho {A : matrix I I ℝ} {B : matrix J J ℝ}
  (ha : A.is_ortho) (hb : B.is_ortho) : (A ⊗ B).is_ortho :=
  by simp [matrix.is_ortho,  transpose_K, K_mul, ha, hb, *] at *
end ortho

section perm
open equiv

variables [decidable_eq I] [decidable_eq J] [mul_zero_one_class α]
variables {A : matrix I I α} {B : matrix J J α}
@[simp] lemma Kronecker.is_perm_of_is_perm (ha : A.is_perm) (hb : B.is_perm) :
(A ⊗ B).is_perm :=
begin
  rcases ha with ⟨σ₁, rfl⟩,
  rcases hb with ⟨σ₂, rfl⟩,
  use prod_congr σ₁ σ₂,
  ext ⟨a,b⟩ ⟨c,d⟩,
  by_cases h1: σ₁ a = c,
  all_goals {simp [*, perm.to_matrix, Kronecker]},
end
end perm

end Kronecker_product

open_locale matrix

section dot_product

lemma dot_product_Kronecker_row [has_mul α] [add_comm_monoid α]
(A : matrix I K α) (B : matrix J L α) (a b : I × J):
dot_product ((A ⊗ B) a) ((A ⊗ B) b) =
∑ (k : K) (l : L), (A a.1 k * B a.2 l) * (A b.1 k * B b.2 l) :=
by simp [dot_product, ←finset.univ_product_univ, finset.sum_product, Kronecker_apply]

lemma dot_product_Kronecker_row' [comm_semiring α]
(A : matrix I K α) (B : matrix J L α) (a b : I × J):
dot_product ((A ⊗ B) a) ((A ⊗ B) b) =
(∑ (k : K), (A a.1 k * A b.1 k)) * ∑ (l : L), (B a.2 l * B b.2 l) :=
```

```
begin
simp [dot_product_Kronecker_row, finset.mul_sum, finset.sum_mul],
repeat {apply finset.sum_congr rfl, intros _ _},
ring
end

lemma dot_product_Kronecker_row_split [comm_semiring α]
(A : matrix I K α) (B : matrix J L α) (a b : I × J):
dot_product ((A ⊗ B) a) ((A ⊗ B) b) =
(dot_product (A a.1)  (A b.1)) * (dot_product (B a.2) (B b.2)) :=
by rw [dot_product_Kronecker_row', dot_product, dot_product]

end dot_product

section sym

/-- `A ⊗ B` is symmetric if `A` and `B` are symmetric. -/
lemma is_sym_K_of [has_mul α] {A : matrix I I α} {B : matrix J J α}
(ha : A.is_sym) (hb : B.is_sym) : (A ⊗ B).is_sym :=
begin
  ext ⟨a, b⟩ ⟨c, d⟩,
  simp [transpose_K, Kronecker, ha.apply', hb.apply'],
end

end sym
/- ## end Kronecker product  -/

end matrix
---------------------------------------------- end of file
```

MAIN2.LEAN

```
/-
Copyright (c) 2021 ***. All rights reserved.
Released under Apache 2.0 license as described in the file LICENSE.
Author: ***.
-/

import finite_field
import circulant_matrix
import diagonal_matrix

/-!
# Hadamard matrices.

This file defines the Hadamard matrices `matrix.Hadamard_matrix` as a type class,
and implements Sylvester's constructions and Payley's constructions of Hadamard matrices and
a Hadamard matrix of order 92.
In particular, this files implements at least one Hadamard matrix of oder `n` for every possible `n ≤ 100`.

## References

*  <https://en.wikipedia.org/wiki/Hadamard_matrix>
*  <https://en.wikipedia.org/wiki/Paley_construction>
* [F.J. MacWilliams, *2 Nonlinear codes, Hadamard matrices, designs and the Golay code*][macwilliams1977]
* [L. D. Baumert, *Discovery of an Hadamard matrix of order 92*][baumert1962]
```

```
## Tags

Hadamard matrix, Hadamard
-/

local attribute [-instance] set.fintype_univ
local attribute [instance] set_fintype

open_locale big_operators

-----------------------------------------------------------------------------
section pre

variables {α β I J : Type*} (S T U : set α)
variables [fintype I] [fintype J]

attribute [simp]
private lemma set.union_to_finset
[decidable_eq α] [fintype ↥S] [fintype ↥T] :
S.to_finset ∪ T.to_finset = (S ∪ T).to_finset :=
(set.to_finset_union S T).symm

@[simp] lemma ite_nested (p : Prop) [decidable p] {a b c d : α}:
ite p (ite p a b) (ite p c d)= ite p a d :=
by by_cases p; simp* at *

@[simp] lemma ite_eq [decidable_eq α] (a x : α) {f : α → β}:
ite (x = a) (f a) (f x)= f x :=
by by_cases x=a; simp* at *

-- The original proof is due to Eric Wieser, given in
-- <https://leanprover.zulipchat.com/#narrow/stream/113489-new-members/topic/card>.
private lemma pick_elements (h : fintype.card I ≥ 3) :
∃ i j k : I, i ≠ j ∧ i ≠ k ∧ j ≠ k :=
begin
  set n := fintype.card I with hn,
  have f := fintype.equiv_fin_of_card_eq hn,
  refine ⟨f.symm ⟨0, by linarith⟩, f.symm ⟨1, by linarith⟩, f.symm ⟨2, by linarith⟩,
    and.imp f.symm.injective.ne (and.imp f.symm.injective.ne f.symm.injective.ne) _⟩,
  dec_trivial,
end
end pre
-----------------------------------------------------------------------------
namespace equiv

variable {I : Type*}

def sum_self_equiv_prod_unit_sum_unit : I ⊕ I ≃  I × (unit ⊕ unit) :=
(equiv.trans (equiv.prod_sum_distrib I unit unit)
             (equiv.sum_congr (equiv.prod_punit I) (equiv.prod_punit I))).symm

@[simp] lemma sum_self_equiv_prod_unit_sum_unit_symm_apply_left (a : unit) (i : I) :
sum_self_equiv_prod_unit_sum_unit.symm (i, sum.inl a) = sum.inl i := rfl

@[simp] lemma sum_self_equiv_prod_unit_sum_unit_symm_apply_right (a : unit) (i : I) :
sum_self_equiv_prod_unit_sum_unit.symm (i, sum.inr a) = sum.inr i := rfl

end equiv
```

```
------------------------------------------------------------------------------
namespace matrix

variables {α β γ I J K L M N: Type*}
variables {R : Type*}
variables {m n: ℕ}
variables [fintype I] [fintype J] [fintype K] [fintype L] [fintype M] [fintype N]
open_locale matrix

section matrix_pre

@[simp] private
lemma push_nag [add_group α] (A : matrix I J α) {i : I} {j : J} {a : α}:
- A i j = a ↔ A i j = -a :=
⟨λ h, eq_neg_of_eq_neg (eq.symm h), λ h, neg_eq_iff_neg_eq.mp (eq.symm h)⟩

lemma dot_product_split_over_subtypes {R} [semiring R]
(v w : I → R) (p : I → Prop) [decidable_pred p] :
dot_product v w =
∑ j : {j : I // p j}, v j * w j + ∑ j : {j : I // ¬ (p j)}, v j * w j :=
by { simp [dot_product], rw fintype.sum_split p}

end matrix_pre

/- ## Hadamard_matrix  -/
section Hadamard_matrix
open fintype finset matrix

class Hadamard_matrix (H : matrix I I ℚ) : Prop :=
(one_or_neg_one []: ∀ i j, (H i j) = 1 ∨ (H i j) = -1)
(orthogonal_rows []: H.has_orthogonal_rows)

-- alternative def
private abbreviation S := {x : ℚ// x = 1 ∨ x = -1}
instance fun_S_to_ℚ: has_coe (β → S) (β → ℚ) := ⟨λ f x, f x⟩
class Hadamard_matrix' (H : matrix I I S):=
(orthogonal_rows []: ∀ i₁ i₂, i₁ ≠ i₂ → dot_product ((H i₁) : (I → ℚ)) (H i₂) = 0)

@[reducible, simp]
def matched (H : matrix I I ℚ) (i₁ i₂ : I) : set I :=
{j : I | H i₁ j = H i₂ j}

@[reducible, simp]
def mismatched (H : matrix I I ℚ) (i₁ i₂ : I) : set I :=
{j : I | H i₁ j ≠ H i₂ j}

section set

/-- `matched H i₁ i₂ ∪ mismatched H i₁ i₂ = I` as sets -/
@[simp] lemma match_union_mismatch (H : matrix I I ℚ) (i₁ i₂ : I) :
matched H i₁ i₂ ∪ mismatched H i₁ i₂ = @set.univ I :=
set.union_compl' _

/-- a variant of `match_union_mismatch` -/
@[simp] lemma match_union_mismatch' (H : matrix I I ℚ) (i₁ i₂ : I) :
{j : I | H i₁ j = H i₂ j} ∪ {j : I | ¬H i₁ j = H i₂ j} = @set.univ I :=
begin
  have h := match_union_mismatch H i₁ i₂,
```

```
  simp* at *,
end

/-- `matched H i₁ i₂ ∪ mismatched H i₁ i₂ = I` as finsets -/
lemma match_union_mismatch_finset [decidable_eq I] (H : matrix I I ℚ) (i₁ i₂ : I) :
(matched H i₁ i₂).to_finset ∪ (mismatched H i₁ i₂).to_finset = @univ I _:=
begin
  simp only [←set.to_finset_union, univ_eq_set_univ_to_finset],
  congr, simp
end

/-- `matched H i₁ i₂` and `mismatched H i₁ i₂` are disjoint as sets -/
@[simp] lemma disjoint_match_mismatch (H : matrix I I ℚ) (i₁ i₂ : I) :
disjoint (matched H i₁ i₂) (mismatched H i₁ i₂) :=
set.disjoint_of_compl' _

/-- `matched H i₁ i₂` and `mismatched H i₁ i₂` are disjoint as finsets -/
@[simp] lemma match_disjoint_mismatch_finset [decidable_eq I] (H : matrix I I ℚ) (i₁ i₂ : I) :
disjoint (matched H i₁ i₂).to_finset (mismatched H i₁ i₂).to_finset :=
by simp [set.to_finset_disjoint_iff]

/-- `|I| = |H.matched i₁ i₂| + |H.mismatched i₁ i₂|`
    for any rows `i₁` `i₂` of a matrix `H` with index type `I`-/
lemma card_match_add_card_mismatch [decidable_eq I] (H : matrix I I ℚ) (i₁ i₂ : I) :
set.card (@set.univ I) = set.card (matched H i₁ i₂) + set.card (mismatched H i₁ i₂) :=
set.card_disjoint_union' (disjoint_match_mismatch _ _ _) (match_union_mismatch _ _ _)

lemma dot_product_split [decidable_eq I] (H : matrix I I ℚ) (i₁ i₂ : I) :
∑ j in (@set.univ I).to_finset, H i₁ j * H i₂ j =
∑ j in (matched H i₁ i₂).to_finset, H i₁ j * H i₂ j +
∑ j in (mismatched H i₁ i₂).to_finset, H i₁ j * H i₂ j :=
set.sum_union' (disjoint_match_mismatch H i₁ i₂) (match_union_mismatch H i₁ i₂)

end set

open matrix Hadamard_matrix

/- ## basic properties  -/
section properties
namespace Hadamard_matrix

variables (H : matrix I I ℚ) [Hadamard_matrix H]

attribute [simp] one_or_neg_one

@[simp] lemma neg_one_or_one (i j : I) : (H i j) = -1 ∨ (H i j) = 1 :=
(one_or_neg_one H i j).swap

@[simp] lemma entry_mul_self (i j : I) :
(H i j) * (H i j) = 1 :=
by rcases one_or_neg_one H i j; simp* at *

variables {H}

lemma entry_eq_one_of_ne_neg_one {i j : I} (h : H i j ≠ -1) :
H i j = 1 := by {have := one_or_neg_one H i j, tauto}

lemma entry_eq_neg_one_of_ne_one {i j : I} (h : H i j ≠ 1) :
```

```
H i j = -1 := by {have := one_or_neg_one H i j, tauto}

lemma entry_eq_neg_one_of {i j k l : I} (h : H i j ≠ H k l) (h' : H i j = 1):
H k l = -1 := by rcases one_or_neg_one H k l; simp* at *

lemma entry_eq_one_of {i j k l : I} (h : H i j ≠ H k l) (h' : H i j = -1):
H k l = 1 := by rcases one_or_neg_one H k l; simp* at *

lemma entry_eq_entry_of {a b c d e f : I} (h₁: H a b ≠ H c d) (h₂: H a b ≠ H e f) :
H c d = H e f :=
begin
  by_cases g : H a b = 1,
  { have g₁ := entry_eq_neg_one_of h₁ g,
    have g₂ := entry_eq_neg_one_of h₂ g,
    linarith },
  { replace g:= entry_eq_neg_one_of_ne_one g,
    have g₁ := entry_eq_one_of h₁ g,
    have g₂ := entry_eq_one_of h₂ g,
    linarith }
end

variables (H)
@[simp] lemma entry_mul_of_ne {i j k l : I} (h : H i j ≠ H k l):
(H i j) * (H k l) = -1 :=
by {rcases one_or_neg_one H i j;
    simp [*, entry_eq_one_of h, entry_eq_neg_one_of h] at *,}

@[simp] lemma row_dot_product_self (i : I) :
dot_product (H i) (H i) = card I := by simp [dot_product, finset.card_univ]

@[simp] lemma col_dot_product_self (j : I) :
dot_product (λ i, H i j) (λ i, H i j) = card I := by simp [dot_product, finset.card_univ]

@[simp] lemma row_dot_product_other {i₁ i₂ : I} (h : i₁ ≠ i₂) :
dot_product (H i₁) (H i₂) = 0 := orthogonal_rows H h

@[simp] lemma row_dot_product_other' {i₁ i₂ : I} (h : i₂ ≠ i₁) :
dot_product (H i₁) (H i₂)= 0 := by simp [ne.symm h]

@[simp] lemma row_dot_product'_other {i₁ i₂ : I} (h : i₁ ≠ i₂) :
∑ j, (H i₁ j) * (H i₂ j) = 0 := orthogonal_rows H h

lemma mul_tanspose [decidable_eq I] :
H ⬝ Hᵀ = (card I : ℚ) • 1 :=
begin
  ext,
  simp [transpose, matrix.mul],
  by_cases i = j; simp [*, mul_one] at *,
end

lemma det_sq [decidable_eq I] :
(det H)^2 = (card I)^(card I) :=
calc (det H)^2 = (det H) * (det H) : by ring
           ... = det (H ⬝ Hᵀ) : by simp
           ... = det ((card I : ℚ) • (1 : matrix I I ℚ)) : by rw mul_tanspose
           ... = (card I : ℚ)^(card I) : by simp

lemma right_invertible [decidable_eq I] :
```

```
H · ((1 / (card I : ℚ)) • Hᵀ) = 1 :=
begin
  have h := mul_tanspose H,
  by_cases hI : card I = 0,
  {exact @eq_of_empty _ _ _ (card_eq_zero_iff.mp hI) _ _}, -- the trivial case
  have hI': (card I : ℚ) ≠ 0, {simp [hI]},
  simp [h, hI'],
end

def invertible [decidable_eq I] : invertible H :=
invertible_of_right_inverse (Hadamard_matrix.right_invertible _)

lemma nonsing_inv_eq [decidable_eq I] : H⁻¹ = (1 / (card I : ℚ)) • Hᵀ :=
inv_eq_right_inv (Hadamard_matrix.right_invertible _)

lemma tanspose_mul [decidable_eq I] :
Hᵀ · H = ((card I) : ℚ) • 1 :=
begin
  rw [←nonsing_inv_right_left (right_invertible H), smul_mul, ←smul_assoc],
  by_cases hI : card I = 0,
  {exact @eq_of_empty _ _ _ (card_eq_zero_iff.mp hI) _ _}, --trivial case
  simp* at *,
end

/-- The dot product of a column with another column equals `0`. -/
@[simp] lemma col_dot_product_other [decidable_eq I] {j₁ j₂ : I} (h : j₁ ≠ j₂) :
dot_product (λ i, H i j₁) (λ i, H i j₂) = 0 :=
begin
  have h':= congr_fun (congr_fun (tanspose_mul H) j₁) j₂,
  simp [matrix.mul, transpose, has_one.one, diagonal, h] at h',
  assumption,
end

/-- The dot product of a column with another column equals `0`. -/
@[simp] lemma col_dot_product_other' [decidable_eq I] {j₁ j₂ : I} (h : j₂ ≠ j₁) :
dot_product (λ i, H i j₁) (λ i, H i j₂)= 0 := by simp [ne.symm h]

/-- Hadamard matrix `H` has orthogonal rows-/
@[simp] lemma has_orthogonal_cols [decidable_eq I] :
H.has_orthogonal_cols:=
by intros i j h; simp [h]

/-- `Hᵀ` is a Hadamard matrix suppose `H` is. -/
instance transpose [decidable_eq I] : Hadamard_matrix Hᵀ :=
begin
  refine{..}, {intros, simp[transpose]},
  simp [transpose_has_orthogonal_rows_iff_has_orthogonal_cols]
end

/-- `Hᵀ` is a Hadamard matrix implies `H` is a Hadamard matrix.-/
lemma of_Hadamard_matrix_transpose [decidable_eq I]
{H : matrix I I ℚ} (h: Hadamard_matrix Hᵀ):
Hadamard_matrix H :=
by convert Hadamard_matrix.transpose Hᵀ; simp

lemma card_match_eq {i₁ i₂ : I} (h: i₁ ≠ i₂):
(set.card (matched H i₁ i₂) : ℚ) = ∑ j in (matched H i₁ i₂).to_finset, H i₁ j * H i₂ j :=
begin
```

```
  simp [matched],
  have h : ∑ (x : I) in {j : I | H i₁ j = H i₂ j}.to_finset, H i₁ x * H i₂ x
         = ∑ (x : I) in {j : I | H i₁ j = H i₂ j}.to_finset, 1,
  { apply finset.sum_congr rfl,
    rintros j hj,
    simp* at * },
  rw [h, ← finset.card_eq_sum_ones_ℚ],
  congr,
end

lemma neg_card_mismatch_eq {i₁ i₂ : I} (h: i₁ ≠ i₂):
- (set.card (mismatched H i₁ i₂) : ℚ) = ∑ j in (mismatched H i₁ i₂).to_finset, H i₁ j * H i₂ j :=
begin
  simp [mismatched],
  have h : ∑ (x : I) in {j : I | H i₁ j ≠ H i₂ j}.to_finset, H i₁ x * H i₂ x
         = ∑ (x : I) in {j : I | H i₁ j ≠ H i₂ j}.to_finset, -1,
  { apply finset.sum_congr rfl, rintros j hj, simp* at * },
  have h' : ∑ (x : I) in {j : I | H i₁ j ≠ H i₂ j}.to_finset, - (1 : ℚ)
          = - ∑ (x : I) in {j : I | H i₁ j ≠ H i₂ j}.to_finset, (1 : ℚ),
  { simp },
  rw [h, h', ← finset.card_eq_sum_ones_ℚ],
  congr,
end

lemma card_mismatch_eq {i₁ i₂ : I} (h: i₁ ≠ i₂):
(set.card (mismatched H i₁ i₂) : ℚ) = - ∑ j in (mismatched H i₁ i₂).to_finset, H i₁ j * H i₂ j :=
by {rw [←neg_card_mismatch_eq]; simp* at *}

/-- `|H.matched i₁ i₂| = |H.mismatched i₁ i₂|` as rational numbers if `H` is a Hadamard matrix.-/
lemma card_match_eq_card_mismatch_ℚ [decidable_eq I] {i₁ i₂ : I} (h: i₁ ≠ i₂):
(set.card (matched H i₁ i₂) : ℚ)= set.card (mismatched H i₁ i₂) :=
begin
  have eq := dot_product_split H i₁ i₂,
  rw [card_match_eq H h, card_mismatch_eq H h],
  simp only [set.to_finset_univ, row_dot_product'_other H h] at eq,
  linarith,
end

/-- `|H.matched i₁ i₂| = |H.mismatched i₁ i₂|` if `H` is a Hadamard matrix.-/
lemma card_match_eq_card_mismatch [decidable_eq I] {i₁ i₂ : I} (h: i₁ ≠ i₂):
set.card (matched H i₁ i₂) = set.card (mismatched H i₁ i₂) :=
by have h := card_match_eq_card_mismatch_ℚ H h; simp * at *

lemma reindex (f : I ≃ J) (g : I ≃ J): Hadamard_matrix (reindex f g H) :=
begin
  refine {..},
  { simp [minor_apply] },
  intros i₁ i₂ h,
  simp [dot_product, minor_apply],
  rw [fintype.sum_equiv (g.symm) _ (λ x, H (f.symm i₁) x * H (f.symm i₂) x) (λ x, rfl)],
  have h' : f.symm i₁ ≠ f.symm i₂, {simp [h]},
  simp [h']
end

end Hadamard_matrix
end properties
/- ## end basic properties  -/
```

```
open Hadamard_matrix

/- ## basic constructions-/
section basic_constr

def H_0 : matrix empty empty ℚ := 1

def H_1 : matrix unit unit ℚ := 1

def H_1' : matrix punit punit ℚ := λ i j, 1

def H_2 : matrix (unit ⊕ unit) (unit ⊕ unit) ℚ :=
(1 :matrix unit unit ℚ).from_blocks 1 1 (-1)

instance Hadamard_matrix.H_0 : Hadamard_matrix H_0 :=
⟨by tidy, by tidy⟩

instance Hadamard_matrix.H_1 : Hadamard_matrix H_1 :=
⟨by tidy, by tidy⟩

instance Hadamard_matrix.H_1' : Hadamard_matrix H_1' :=
⟨by tidy, by tidy⟩

instance Hadamard_matrix.H_2 : Hadamard_matrix H_2 :=
⟨ by tidy,
  λ i₁ i₂ h, by { cases i₁, any_goals {cases i₂},
                  any_goals {simp[*, H_2, dot_product, fintype.sum_sum_type] at *} }
⟩

end basic_constr
/- ## end basic constructions-/

/- ## "normalize" constructions-/
section normalize

open matrix Hadamard_matrix

/-- negate row `i` of matrix `A`; `[decidable_eq I]` is required for `update_row` -/
def neg_row [has_neg α] [decidable_eq I] (A : matrix I J α) (i : I) :=
update_row A i (- A i)

/-- negate column `j` of matrix `A`; `[decidable_eq J]` is required for `update_column` -/
def neg_col [has_neg α] [decidable_eq J] (A : matrix I J α) (j : J) :=
update_column A j (-λ i, A i j)

section neg

/-- Negating row `i` and then column `j` equals negating column `j` first and then row `i`. -/
lemma neg_row_neg_col_comm [has_neg α] [decidable_eq I] [decidable_eq J]
(A : matrix I J α) (i : I) (j : J) :
(A.neg_row i).neg_col j = (A.neg_col j).neg_row i :=
begin
  ext a b,
  simp [neg_row, neg_col, update_column_apply, update_row_apply],
  by_cases a = i,
  any_goals {by_cases b = j},
  any_goals {simp* at *},
end
```

```
lemma transpose_neg_row [has_neg α] [decidable_eq I] (A : matrix I J α) (i : I) :
(A.neg_row i)ᵀ = Aᵀ.neg_col i :=
by simp [← update_column_transpose, neg_row, neg_col]

lemma transpose_neg_col [has_neg α] [decidable_eq J] (A : matrix I J α) (j : J) :
(A.neg_col j)ᵀ = Aᵀ.neg_row j :=
by {simp [← update_row_transpose, neg_row, neg_col, trans_row_eq_col]}

lemma neg_row_add [add_comm_group α] [decidable_eq I]
(A B : matrix I J α) (i : I) :
(A.neg_row i) + (B.neg_row i) = (A + B).neg_row i :=
begin
  ext a b,
  simp [neg_row, neg_col, update_column_apply, update_row_apply],
  by_cases a = i,
  any_goals {simp* at *},
  abel
end

lemma neg_col_add [add_comm_group α] [decidable_eq J]
(A B : matrix I J α) (j : J) :
(A.neg_col j) + (B.neg_col j) = (A + B).neg_col j :=
begin
  ext a b,
  simp [neg_row, neg_col, update_column_apply, update_row_apply],
  by_cases b = j,
  any_goals {simp* at *},
  abel
end

/-- Negating the same row and column of diagonal matrix `A` equals `A` itself. -/
lemma neg_row_neg_col_eq_self_of_is_diag [add_group α] [decidable_eq I]
{A : matrix I I α} (h : A.is_diagonal) (i : I) :
(A.neg_row i).neg_col i = A :=
begin
  ext a b,
  simp [neg_row, neg_col, update_column_apply, update_row_apply],
  by_cases h₁ : a = i,
  any_goals {by_cases h₂ : b = i},
  any_goals {simp* at *},
  { simp [h.apply_ne' h₂] },
  { simp [h.apply_ne h₁] },
end

end neg

variables [decidable_eq I] (H : matrix I I ℚ) [Hadamard_matrix H]

/-- Negating any row `i` of a Hadamard matrix `H` produces another Hadamard matrix. -/
instance Hadamard_matrix.neg_row (i : I) :
Hadamard_matrix (H.neg_row i) :=
begin
  -- first goal
  refine {..},
  { intros j k,
    simp [neg_row,  update_row_apply],
    by_cases j = i; simp* at * },
  -- second goal
```

```
  { intros j k hjk,
    by_cases h1 : j = i, any_goals {by_cases h2 : k = i},
    any_goals {simp [*, neg_row, update_row_apply]},
    tidy }
end

/-- Negating any column `j` of a Hadamard matrix `H` produces another Hadamard matrix. -/
instance Hadamard_matrix.neg_col (j : I) :
Hadamard_matrix (H.neg_col j) :=
begin
  apply of_Hadamard_matrix_transpose, --changes the goal to `(H.neg_col j)ᵀ.Hadamard_matrix`
  simp [transpose_neg_col, Hadamard_matrix.neg_row]
  -- `(H.neg_col j)ᵀ = Hᵀ.neg_row j`, in which the RHS has been proved to be a Hadamard matrix.
end

end normalize
/- ## end "normalize" constructions -/


/- ## special cases -/
section special_cases

namespace Hadamard_matrix
variables (H : matrix I I ℚ) [Hadamard_matrix H]

/-- normalized Hadamard matrix -/
def is_normalized [inhabited I] : Prop :=
H (default I) = 1 ∧ (λ i, H i (default I)) = 1

/-- skew Hadamard matrix -/
def is_skew [decidable_eq I] : Prop :=
Hᵀ + H = 2

/-- regular Hadamard matrix -/
def is_regular : Prop :=
∀ i j, ∑ b, H i b = ∑ a, H a j

variable {H}

lemma is_skew.eq [decidable_eq I] (h : is_skew H) :
Hᵀ + H = 2 := h

@[simp] lemma is_skew.apply_eq
[decidable_eq I] (h : is_skew H) (i : I) :
H i i + H i i = 2 :=
by replace h:= congr_fun (congr_fun h i) i; simp * at *

@[simp] lemma is_skew.apply_ne
[decidable_eq I] (h : is_skew H) {i j : I} (hij : i ≠ j) :
H j i + H i j = 0 :=
by replace h:= congr_fun (congr_fun h i) j; simp * at *

lemma is_skew.of_neg_col_row_of_is_skew
[decidable_eq I] (i : I) (h : Hadamard_matrix.is_skew H) :
is_skew ((H.neg_row i).neg_col i) :=
begin
  simp [is_skew],
  -- to show ((H.neg_row i).neg_col i)ᵀ + (H.neg_row i).neg_col i = 2
```

```
  nth_rewrite 0 [neg_row_neg_col_comm],
  simp [transpose_neg_row, transpose_neg_col, neg_row_add, neg_col_add],
  rw [h.eq],
  convert neg_row_neg_col_eq_self_of_is_diag _ _,
  apply is_diagonal_add; by simp
end


end Hadamard_matrix

end special_cases
/- ## end special cases -/

/- ## Sylvester construction  -/
section Sylvester_constr

def Sylvester_constr₀ (H : matrix I I ℚ) [Hadamard_matrix H] : matrix (I ⊕ I) (I ⊕ I) ℚ :=
H.from_blocks H H (-H)

@[instance]
theorem Hadamard_matrix.Sylvester_constr₀ (H : matrix I I ℚ) [Hadamard_matrix H] :
Hadamard_matrix (matrix.Sylvester_constr₀ H) :=
begin
  refine{..},
  { rintros (i | i)  (j | j);
    simp [matrix.Sylvester_constr₀] },
  rintros (i | i) (j | j) h,
  all_goals {simp [matrix.Sylvester_constr₀, dot_product_block', *]},
  any_goals {rw [← dot_product], have h' : i ≠ j; simp* at *}
end

def Sylvester_constr₀' (H : matrix I I ℚ) [Hadamard_matrix H]:
matrix (I × (unit ⊕ unit)) (I × (unit ⊕ unit)) ℚ :=
H ⊗ H_2

local notation `reindex_map` := equiv.sum_self_equiv_prod_unit_sum_unit

lemma Sylvester_constr₀'_eq_reindex_Sylvester_constr₀
(H : matrix I I ℚ) [Hadamard_matrix H] :
H.Sylvester_constr₀' = reindex reindex_map reindex_map H.Sylvester_constr₀:=
begin
  ext ⟨i, a⟩ ⟨j, b⟩,
  simp [Sylvester_constr₀', Sylvester_constr₀, Kronecker, H_2, from_blocks],
  rcases a with (a | a),
  any_goals {rcases b with (b | b)},
  any_goals {simp [one_apply]},
end

@[instance]
theorem Hadamard_matrix.Sylvester_constr₀' (H : matrix I I ℚ) [Hadamard_matrix H] :
Hadamard_matrix (Sylvester_constr₀' H) :=
begin
  convert Hadamard_matrix.reindex H.Sylvester_constr₀ reindex_map reindex_map,
  exact H.Sylvester_constr₀'_eq_reindex_Sylvester_constr₀,
end

theorem Hadamard_matrix.order_conclusion_1:
∀ (n : ℕ),  ∃ {I : Type*} [inst : fintype I]
```

```
(H : @matrix I I inst inst ℚ) [@Hadamard_matrix I inst H],
@fintype.card I inst = 2^n :=
begin
  intro n,
  induction n with n ih,
  -- the case 0
  {exact ⟨punit, infer_instance, H_1', infer_instance, by simp⟩},
  -- the case n.succ
  rcases ih with ⟨I, inst, H, h, hI⟩, resetI, -- unfold the IH
  refine ⟨I ⊕ I, infer_instance, H.Sylvester_constr₀, infer_instance, _⟩,
  rw [fintype.card_sum, hI], ring_nf, -- this line proves `card (I ⊕ I) = 2 ^ n.succ`
end

end Sylvester_constr
/- ## end Sylvester construction  -/

/- ## general Sylvester construction  -/
section general_Sylvester_constr

def Sylvester_constr
(H₁ : matrix I I ℚ) [Hadamard_matrix H₁] (H₂ : matrix J J ℚ) [Hadamard_matrix H₂] :
matrix (I × J) (I × J) ℚ := H₁ ⊗ H₂

@[instance] theorem Hadamard_matrix.Sylvester_constr'
(H₁ : matrix I I ℚ) [Hadamard_matrix H₁] (H₂ : matrix J J ℚ) [Hadamard_matrix H₂] :
Hadamard_matrix (H₁ ⊗ H₂) :=
begin
  refine {..},
  -- first goal
  { rintros ⟨i₁, j₁⟩ ⟨i₂, j₂⟩,
    simp [Kronecker],
    -- the current goal : H₁ i₁ i₂ * H₂ j₁ j₂ = 1 ∨ H₁ i₁ i₂ * H₂ j₁ j₂ = -1
    obtain (h | h) := one_or_neg_one H₁ i₁ i₂; -- prove by cases : H₁ i₁ i₂ = 1 or -1
    simp [h] },
  -- second goal
  rintros ⟨i₁, j₁⟩ ⟨i₂, j₂⟩ h,
  simp [dot_product_Kronecker_row_split],
  -- by cases j₁ = j₂; simp* closes the case j₁ ≠ j₂
  by_cases hi: i₁ = i₂, any_goals {simp*},
  -- the left case: i₁ = i₂
  by_cases hi: j₁ = j₂, any_goals {simp* at *},
end

/-- wraps `Hadamard_matrix.Sylvester_constr'`-/
@[instance] theorem Hadamard_matrix.Sylvester_constr
(H₁ : matrix I I ℚ) [Hadamard_matrix H₁] (H₂ : matrix J J ℚ) [Hadamard_matrix H₂] :
Hadamard_matrix (Sylvester_constr H₁ H₂) :=
Hadamard_matrix.Sylvester_constr' H₁ H₂

theorem {u v} Hadamard_matrix.order_conclusion_2 {I : Type u} {J : Type v} [fintype I] [fintype J]
(H₁ : matrix I I ℚ) [Hadamard_matrix H₁] (H₂ : matrix J J ℚ) [Hadamard_matrix H₂] :
∃ {K : Type max u v} [inst : fintype K] (H : @matrix K K inst inst ℚ),
by exactI Hadamard_matrix H ∧ card K = card I * card J :=
⟨(I × J), _, Sylvester_constr H₁ H₂, ⟨infer_instance, card_prod I J⟩⟩

end general_Sylvester_constr
/- ## end general Sylvester construction  -/
```

```
/- ## Paley construction -/
section Paley_construction

variables {F : Type*} [field F] [fintype F] [decidable_eq F] {p : ℕ} [char_p F p]
local notation `q` := fintype.card F

open finite_field

/- ## Jacobsthal_matrix -/

variable (F) -- `F` is an explicit variable to `Jacobsthal_matrix`.

@[reducible] def Jacobsthal_matrix : matrix F F ℚ := λ a b, χ (a-b)
-- We will use `J` to denote `Jacobsthal_matrix F` in annotations.

namespace Jacobsthal_matrix

/-- `J` is the circulant matrix `cir χ`. -/
lemma eq_cir : (Jacobsthal_matrix F) = cir χ := rfl

variable {F} -- this line makes `F` an implicit variable to the following lemmas/defs

@[simp] lemma diag_entry_eq_zero (i : F) : (Jacobsthal_matrix F) i i = 0 :=
by simp [Jacobsthal_matrix]

@[simp] lemma non_diag_entry_eq {i j : F} (h : i ≠ j):
(Jacobsthal_matrix F) i j = 1 ∨ (Jacobsthal_matrix F) i j = -1 :=
by simp [*, Jacobsthal_matrix]

@[simp] lemma non_diag_entry_Euare_eq {i j : F} (h : i ≠ j):
(Jacobsthal_matrix F) i j * (Jacobsthal_matrix F) i j = 1 :=
by obtain (h₁ | h₂) := Jacobsthal_matrix.non_diag_entry_eq h; simp*

@[simp] lemma entry_Euare_eq (i j : F) :
(Jacobsthal_matrix F) i j * (Jacobsthal_matrix F) i j = ite (i=j) 0 1 :=
by by_cases i=j; simp * at *

-- JJᵀ = qI - 𝟙
lemma mul_transpose_self (hp : p ≠ 2) :
(Jacobsthal_matrix F) ⬝ (Jacobsthal_matrix F)ᵀ = (q : ℚ) • 1 - 𝟙 :=
begin
  ext i j,
  simp [mul_apply, all_one, Jacobsthal_matrix, one_apply],
  -- the current goal is
  -- ∑ (x : F), χ (i - x) * χ (j - x) = ite (i = j) q 0 - 1
  by_cases i = j,
  -- when i = j
  { simp[h, sum_ite, filter_ne, fintype.card],
    rw [@card_erase_of_mem' _ _ j (@finset.univ F _) _];
    simp },
  -- when i ≠ j
  simp [quad_char.sum_mul h hp, h],
end

-- J ⬝ 𝟙 = 0
@[simp] lemma mul_all_one (hp : p ≠ 2) :
(Jacobsthal_matrix F) ⬝ (𝟙 : matrix F F ℚ) = 0 :=
begin
```

```
  ext i j,
  simp [all_one, Jacobsthal_matrix, mul_apply],
  -- the current goal: ∑ (x : F), χ (i - x) = 0
  exact quad_char.sum_eq_zero_reindex_1 hp,
end

-- 𝟙 · J = 0
@[simp] lemma all_one_mul (hp : p ≠ 2) :
(𝟙 : matrix F F ℚ) · (Jacobsthal_matrix F) = 0 :=
begin
  ext i j,
  simp [all_one, Jacobsthal_matrix, mul_apply],
  exact quad_char.sum_eq_zero_reindex_2 hp,
end

-- J · col 1 = 0
@[simp] lemma mul_col_one (hp : p ≠ 2) :
Jacobsthal_matrix F · col 1 = 0 :=
begin
  ext,
  simp [Jacobsthal_matrix, mul_apply],
  -- the current goal: ∑ (x : F), χ (i - x) = 0
  exact quad_char.sum_eq_zero_reindex_1 hp,
end

-- row 1 · Jᵀ = 0
@[simp] lemma row_one_mul_transpose (hp : p ≠ 2) :
row 1 · (Jacobsthal_matrix F)ᵀ = 0 :=
begin
  apply eq_of_transpose_eq,
  simp,
  exact mul_col_one hp
end

variables {F}

lemma is_sym_of (h : q ≡ 1 [MOD 4]) :
(Jacobsthal_matrix F).is_sym :=
by ext; simp [Jacobsthal_matrix, quad_char_is_sym_of' h i j]

lemma is_skewsym_of (h : q ≡ 3 [MOD 4]) :
(Jacobsthal_matrix F).is_skewsym :=
by ext; simp [Jacobsthal_matrix, quad_char_is_skewsym_of' h i j]

lemma is_skesym_of' (h : q ≡ 3 [MOD 4]) :
(Jacobsthal_matrix F)ᵀ = - (Jacobsthal_matrix F) :=
begin
  have := Jacobsthal_matrix.is_skewsym_of h,
  unfold matrix.is_skewsym at this,
  nth_rewrite 1 [← this],
  simp,
end

end Jacobsthal_matrix
/- ## end Jacobsthal_matrix -/

open Jacobsthal_matrix
```

```
/- ## Paley_constr_1 -/

variable (F)
def Paley_constr_1 : matrix (unit ⊕ F) (unit ⊕ F) ℚ :=
(1 : matrix unit unit ℚ).from_blocks (- row 1) (col 1) (1 + (Jacobsthal_matrix F))

@[simp] def Paley_constr_1'_aux : matrix (unit ⊕ F) (unit ⊕ F) ℚ :=
(0 : matrix unit unit ℚ).from_blocks (- row 1) (col 1) (Jacobsthal_matrix F)

def Paley_constr_1' := 1 + (Paley_constr_1'_aux F)

lemma Paley_constr_1'_eq_Paley_constr_1 :
Paley_constr_1' F = Paley_constr_1 F :=
begin
  simp only [Paley_constr_1', Paley_constr_1'_aux, Paley_constr_1, ←from_blocks_one, from_blocks_add],
  simp,
end

variable {F}

/-- if `q ≡ 3 [MOD 4]`, `Paley_constr_1 F` is a Hadamard matrix. -/
@[instance]
theorem Hadamard_matrix.Paley_constr_1 (h : q ≡ 3 [MOD 4]):
Hadamard_matrix (Paley_constr_1 F) :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- derive the char p of F
  resetI, -- resets the instance cache
  obtain ⟨hp, h'⟩ := char_ne_two_of' p h, -- prove p ≠ 2
  refine {..},
  -- first goal
  {
    rintros (i | i)  (j | j),
    all_goals {simp [Paley_constr_1, one_apply, Jacobsthal_matrix]},
    {by_cases i = j; simp*}
  },
  -- second goal
  rw ←mul_tranpose_is_diagonal_iff_has_orthogonal_rows,   -- changes the goal to prove J · Jᵀ is diagonal
  simp [Paley_constr_1, from_blocks_transpose, from_blocks_multiply,
        matrix.add_mul, matrix.mul_add, col_one_mul_row_one],
  rw [mul_col_one hp, row_one_mul_transpose hp, mul_transpose_self hp],
  simp,
  convert is_diagnoal_of_block_conditions ⟨is_diagonal_of_unit _, _, rfl, rfl⟩,
  -- to show the lower right corner block is diagonal
  {rw [is_skesym_of' h, add_assoc, add_comm, add_assoc], simp},
  any_goals {assumption},
end

open Hadamard_matrix

/-- if `q ≡ 3 [MOD 4]`, `Paley_constr_1 F` is a skew Hadamard matrix. -/
theorem Hadamard_matrix.Paley_constr_1_is_skew (h : q ≡ 3 [MOD 4]):
@is_skew _ _ (Paley_constr_1 F) (Hadamard_matrix.Paley_constr_1 h) _ :=
begin
  simp [is_skew, Paley_constr_1, from_blocks_transpose,
        from_blocks_add, is_skesym_of' h],
  have : 1 + -Jacobsthal_matrix F + (1 + Jacobsthal_matrix F) = 1 + 1,
  {noncomm_ring},
  rw [this], clear this,
```

```
  ext (a | i) (b | j),
  swap 3, rintro (b | j),
  any_goals {simp [one_apply, from_blocks, bit0]},
end

/- ## end Paley_constr_1 -/

/- ## Paley_constr_2 -/

/- # Paley_constr_2_helper -/
namespace Paley_constr_2

variable (F)

def C : matrix (unit ⊕ unit) (unit ⊕ unit) ℚ :=
(1 : matrix unit unit ℚ).from_blocks (-1) (-1) (-1)

/-- C is symmetric. -/
@[simp] lemma C_is_sym : C.is_sym :=
is_sym_of_block_conditions ⟨by simp, by simp, by simp⟩

def D : matrix (unit ⊕ unit) (unit ⊕ unit) ℚ :=
(1 : matrix unit unit ℚ).from_blocks 1 1 (-1)

/-- D is symmetric. -/
@[simp] lemma D_is_sym : D.is_sym :=
is_sym_of_block_conditions ⟨by simp, by simp, by simp⟩

/-- C · D = - D · C -/
lemma C_mul_D_anticomm : C · D = - D · C :=
begin
  ext (i | i) (j | j),
  swap 3, rintros (j | j),
  any_goals {simp [from_blocks_multiply, C, D]}
end

def E : matrix (unit ⊕ unit) (unit ⊕ unit) ℚ :=
(2 : matrix unit unit ℚ).from_blocks 0 0 2

/-- E is diagonal. -/
@[simp] lemma E_is_diagonal : E.is_diagonal :=
is_diagnoal_of_block_conditions ⟨by simp, by simp, rfl, rfl⟩

/-- C · C = E -/
@[simp] lemma C_mul_self : C · C = E :=
by simp [from_blocks_transpose, from_blocks_multiply, E, C]; congr' 1

/-- C · Cᵀ = E -/
@[simp] lemma C_mul_transpose_self : C · Cᵀ = E :=
by simp [C_is_sym.eq]

/-- D · D = E -/
@[simp] lemma D_mul_self : D · D = E :=
by simp [from_blocks_transpose, from_blocks_multiply, E, D]; congr' 1

/-- D · Dᵀ = E -/
@[simp] lemma D_mul_transpose_self : D · Dᵀ = E :=
by simp [D_is_sym.eq]
```

```
def replace (A : matrix I J ℚ) :
matrix (I × (unit ⊕ unit)) (J × (unit ⊕ unit)) ℚ :=
λ ⟨i, a⟩ ⟨j, b⟩,
if (A i j = 0)
then C a b
else (A i j) • D a b

variable (F)

/-- `(replace A)ᵀ = replace (Aᵀ)` -/
lemma transpose_replace (A : matrix I J ℚ) :
(replace A)ᵀ = replace (Aᵀ) :=
begin
  ext ⟨i, a⟩ ⟨j, b⟩,
  simp [transpose_apply, replace],
  congr' 1,
  {rw [C_is_sym.apply']},
  {rw [D_is_sym.apply']},
end

variable (F)

/-- `replace A` is a symmetric matrix if `A` is. -/
lemma replace_is_sym_of {A : matrix I I ℚ} (h : A.is_sym) :
(replace A).is_sym:=
begin
  ext ⟨i, a⟩ ⟨j, b⟩,
  simp [transpose_replace, replace, h.apply', C_is_sym.apply', D_is_sym.apply']
end

/-- `replace 0 = I ⊗ C` -/
lemma replace_zero :
replace (0 : matrix unit unit ℚ) = 1 ⊗ C :=
begin
  ext ⟨a, b⟩ ⟨c, d⟩,
  simp [replace, Kronecker, one_apply]
end

/-- `replace A = A ⊗ D` for a matrix `A` with no `0` entries. -/
lemma replace_matrix_of_no_zero_entry
{A : matrix I J ℚ} (h : ∀ i j, A i j ≠ 0) : replace A = A ⊗ D :=
begin
  ext ⟨i, a⟩ ⟨j, b⟩,
  simp [replace, Kronecker],
  intro g,
  exact absurd g (h i j)
end

/-- In particular, we can apply `replace_matrix_of_no_zero_entry` to `- row 1`. -/
lemma replace_neg_row_one :
replace (-row 1 : matrix unit F ℚ) = (-row 1) ⊗ D :=
replace_matrix_of_no_zero_entry (λ a i, by simp [row])

/-- `replace J = J ⊗ D + I ⊗ C` -/
lemma replace_Jacobsthal :
replace (Jacobsthal_matrix F) =
(Jacobsthal_matrix F) ⊗ D + 1 ⊗ C:=
begin
```

```
  ext ⟨i, a⟩ ⟨j, b⟩,
  by_cases i = j, --inspect the diagonal and non-diagonal entries respectively
  any_goals {simp [h, Jacobsthal_matrix, replace, Kronecker]},
end

/-- `(replace 0) · (replace 0)ᵀ= I ⊗ E` -/
@[simp] lemma replace_zero_mul_transpose_self :
replace (0 : matrix unit unit ℚ) · (replace (0 : matrix unit unit ℚ))ᵀ = 1 ⊗ E :=
by simp [replace_zero, transpose_K, K_mul]

/-- `(replace A) · (replace A)ᵀ = (A · Aᵀ) ⊗ E` -/
@[simp] lemma replace_matrix_of_no_zero_entry_mul_transpose_self
{A : matrix I J ℚ} (h : ∀ i j, A i j ≠ 0) :
(replace A) · (replace A)ᵀ = (A · Aᵀ) ⊗ E :=
by simp [replace_matrix_of_no_zero_entry h, transpose_K, K_mul]

variable {F}

lemma replace_Jacobsthal_mul_transpose_self' (h : q ≡ 1 [MOD 4]) :
replace (Jacobsthal_matrix F) · (replace (Jacobsthal_matrix F))ᵀ =
((Jacobsthal_matrix F) · (Jacobsthal_matrix F)ᵀ + 1) ⊗ E :=
begin
  simp [transpose_replace, (is_sym_of h).eq],
  simp [replace_Jacobsthal, matrix.add_mul, matrix.mul_add,
        K_mul, C_mul_D_anticomm, add_K],
  noncomm_ring
end

/-- enclose `replace_Jacobsthal_mul_transpose_self'` by replacing `J · Jᵀ` with `qI - 𝟙` -/
@[simp]lemma replace_Jacobsthal_mul_transpose_self (h : q ≡ 1 [MOD 4]) :
replace (Jacobsthal_matrix F) · (replace (Jacobsthal_matrix F))ᵀ =
(((q : ℚ) + 1) • (1 : matrix F F ℚ) - 𝟙) ⊗ E :=
begin
  obtain ⟨p, inst⟩ := char_p.exists F, -- obtains the character p of F
  resetI, -- resets the instance cache
  obtain hp := char_ne_two_of p (or.inl h),  -- hp: p ≠ 2
  simp [replace_Jacobsthal_mul_transpose_self' h, add_smul],
  rw [mul_transpose_self hp],
  congr' 1, noncomm_ring,
  assumption
end

end Paley_constr_2
/- # end Paley_constr_2_helper -/

open Paley_constr_2

variable (F)
def Paley_constr_2 :=
(replace (0 : matrix unit unit ℚ)).from_blocks
(replace (- row 1))
(replace (- col 1))
(replace (Jacobsthal_matrix F))

variable {F}
/-- `Paley_constr_2 F` is a symmetric matrix when `card F ≡ 1 [MOD 4]`. -/
@[simp]
lemma Paley_constr_2_is_sym (h : q ≡ 1 [MOD 4]) :
```

```
(Paley_constr_2 F).is_sym :=
begin
  convert is_sym_of_block_conditions ⟨_, _, _⟩,
  { simp [replace_zero] }, -- `0` is symmetric
  { apply replace_is_sym_of (is_sym_of h) }, -- `J` is symmetric
  { simp [transpose_replace] } -- `(replace (-row 1))ᵀ = replace (-col 1)`
end

variable (F)
/-- Every entry of `Paley_constr_2 F` equals `1` or `-1`. -/
lemma Paley_constr_2.one_or_neg_one :
∀ (i j : unit × (unit ⊕ unit) ⊕ F × (unit ⊕ unit)),
Paley_constr_2 F i j = 1 ∨ Paley_constr_2 F i j = -1 :=
begin
  rintros (⟨a, (u₁|u₂)⟩ | ⟨i, (u₁ | u₂)⟩) (⟨b, (u₃|u₄)⟩ | ⟨j, (u₃ | u₄)⟩),
  all_goals {simp [Paley_constr_2, one_apply, Jacobsthal_matrix, replace, C, D]},
  all_goals {by_cases i = j},
  any_goals {simp [h]},
end

variable {F}

@[instance]
theorem Hadamard_matrix.Paley_constr_2 (h : q ≡ 1 [MOD 4]):
Hadamard_matrix (Paley_constr_2 F) :=
begin
  refine {..},
  -- the first goal
  { exact Paley_constr_2.one_or_neg_one F },
  -- the second goal
  -- turns the goal to `Paley_constr_2 F · (Paley_constr_2 F)ᵀ` is diagonal
  rw ←mul_tranpose_is_diagonal_iff_has_orthogonal_rows,
  -- sym : `Paley_constr_2 F · (Paley_constr_2 F)ᵀ` is symmetric
  have sym := mul_transpose_self_is_sym (Paley_constr_2 F),
  -- The next `simp` turns `Paley_constr_2 F · (Paley_constr_2 F)ᵀ` into a block form.
  simp [Paley_constr_2, from_blocks_transpose, from_blocks_multiply] at *,
  convert is_diagnoal_of_sym_block_conditions sym ⟨_, _, _⟩, -- splits into the three goals
  any_goals {clear sym},
  -- to prove the upper left corner block is diagonal.
  { simp [row_one_mul_col_one, ← add_K],
    apply K_is_diagonal_of; simp },
  -- to prove the lower right corner block is diagonal.
  { simp [h, col_one_mul_row_one, ← add_K],
    apply smul_is_diagonal_of,
    apply K_is_diagonal_of; simp },
  -- to prove the upper right corner block is `0`.
  { obtain ⟨p, inst⟩ := char_p.exists F, -- obtains the character p of F
    resetI, -- resets the instance cache
    obtain hp := char_ne_two_of p (or.inl h), -- hp: p ≠ 2
    simp [transpose_replace, (is_sym_of h).eq],
    simp [replace_zero, replace_neg_row_one, replace_Jacobsthal,
          matrix.mul_add, K_mul, C_mul_D_anticomm],
    rw [←(is_sym_of h).eq, row_one_mul_transpose hp],
    simp, assumption }
end

/- ## end Paley_constr_2 -/
end Paley_construction
```

```
/- ## end Paley construction -/


/- ## order 92-/
section order_92

namespace H_92

def a : fin 23 → ℚ :=
![ 1,  1, -1, -1, -1,  1, -1, -1, -1,  1, -1,  1,  1, -1,  1, -1, -1, -1,  1, -1, -1, -1,  1]
def b : fin 23 → ℚ :=
![ 1, -1,  1,  1, -1,  1,  1, -1, -1,  1,  1,  1,  1,  1,  1, -1, -1,  1,  1, -1,  1,  1, -1]
def c : fin 23 → ℚ :=
![ 1,  1,  1, -1, -1, -1,  1,  1, -1,  1, -1,  1,  1, -1,  1, -1,  1,  1, -1, -1, -1,  1,  1]
def d : fin 23 → ℚ :=
![ 1,  1,  1, -1,  1,  1,  1, -1,  1, -1, -1, -1, -1, -1, -1,  1, -1,  1,  1,  1, -1,  1,  1]

abbreviation A := cir a
abbreviation B := cir b
abbreviation C := cir c
abbreviation D := cir d

@[simp] lemma a.one_or_neg_one : ∀ i, a i ∈ ({1, -1} : set ℚ) :=
λ i, begin simp, dec_trivial! end -- `dec_trivial!` inspects every entry
@[simp] lemma b.one_or_neg_one : ∀ i, b i ∈ ({1, -1} : set ℚ) :=
λ i, begin simp, dec_trivial! end
@[simp] lemma c.one_or_neg_one : ∀ i, c i ∈ ({1, -1} : set ℚ) :=
λ i, begin simp, dec_trivial! end
@[simp] lemma d.one_or_neg_one : ∀ i, d i ∈ ({1, -1} : set ℚ) :=
λ i, begin simp, dec_trivial! end

@[simp] lemma A.one_or_neg_one : ∀ i j, A i j = 1 ∨ A i j = -1 :=
by convert cir_entry_in_of_vec_entry_in a.one_or_neg_one
@[simp] lemma A.neg_one_or_one : ∀ i j, A i j = -1 ∨ A i j = 1 :=
λ i j, (A.one_or_neg_one i j).swap
@[simp] lemma B.one_or_neg_one : ∀ i j, B i j = 1 ∨ B i j = -1 :=
by convert cir_entry_in_of_vec_entry_in b.one_or_neg_one
@[simp] lemma B.neg_one_or_one : ∀ i j, B i j = -1 ∨ B i j = 1 :=
λ i j, (B.one_or_neg_one i j).swap
@[simp] lemma C.one_or_neg_one : ∀ i j, C i j = 1 ∨ C i j = -1 :=
by convert cir_entry_in_of_vec_entry_in c.one_or_neg_one
@[simp] lemma C.neg_one_or_one : ∀ i j, C i j = -1 ∨ C i j = 1 :=
λ i j, (C.one_or_neg_one i j).swap
@[simp] lemma D.one_or_neg_one : ∀ i j, D i j = 1 ∨ D i j = -1 :=
by convert cir_entry_in_of_vec_entry_in d.one_or_neg_one
@[simp] lemma D.neg_one_or_one : ∀ i j, D i j = -1 ∨ D i j = 1 :=
λ i j, (D.one_or_neg_one i j).swap

@[simp] lemma a_is_sym : ∀ (i : fin 23), a (-i) = a i := by dec_trivial

@[simp] lemma a_is_sym' : ∀ (i : fin 23),
![(1 : ℚ), 1, -1, -1, -1,  1, -1, -1, -1,  1, -1,  1,  1, -1,  1, -1, -1, -1,  1, -1, -1, -1,  1] (-i) =
![(1 : ℚ), 1, -1, -1, -1,  1, -1, -1, -1,  1, -1,  1,  1, -1,  1, -1, -1, -1,  1, -1, -1, -1,  1]   i :=
by convert a_is_sym

@[simp] lemma b_is_sym : ∀ (i : fin 23), b (-i) = b i := by dec_trivial

@[simp] lemma b_is_sym' : ∀ (i : fin 23),
```

```
![(1 : ℚ), -1,  1,  1, -1,  1,  1, -1, -1,  1,  1,  1,  1,  1,  1, -1, -1,  1,  1, -1,  1,  1, -1] (-i) =
![(1 : ℚ), -1,  1,  1, -1,  1,  1, -1, -1,  1,  1,  1,  1,  1,  1, -1, -1,  1,  1, -1,  1,  1, -1]   i :=
by convert b_is_sym

@[simp] lemma c_is_sym : ∀ (i : fin 23), c (-i) = c i := by dec_trivial

@[simp] lemma c_is_sym' : ∀ (i : fin 23),
![ (1 : ℚ), 1,  1, -1, -1, -1,  1,  1, -1,  1, -1,  1,  1, -1,  1, -1,  1,  1, -1, -1, -1,  1,  1] (-i) =
![ (1 : ℚ), 1,  1, -1, -1, -1,  1,  1, -1,  1, -1,  1,  1, -1,  1, -1,  1,  1, -1, -1, -1,  1,  1]   i :=
by convert c_is_sym

@[simp] lemma d_is_sym : ∀ (i : fin 23), d (-i) = d i := by dec_trivial

@[simp] lemma d_is_sym' : ∀ (i : fin 23),
![ (1 : ℚ), 1,  1, -1,  1,  1,  1, -1,  1, -1, -1, -1, -1, -1, -1,  1, -1,  1,  1,  1, -1,  1,  1] (-i) =
![ (1 : ℚ), 1,  1, -1,  1,  1,  1, -1,  1, -1, -1, -1, -1, -1, -1,  1, -1,  1,  1,  1, -1,  1,  1]   i :=
by convert d_is_sym

@[simp] lemma A_is_sym : Aᵀ = A :=
by rw [←is_sym, cir_is_sym_ext_iff]; exact a_is_sym
@[simp] lemma B_is_sym : Bᵀ = B :=
by rw [←is_sym, cir_is_sym_ext_iff]; exact b_is_sym
@[simp] lemma C_is_sym : Cᵀ = C :=
by rw [←is_sym, cir_is_sym_ext_iff]; exact c_is_sym
@[simp] lemma D_is_sym : Dᵀ = D :=
by rw [←is_sym, cir_is_sym_ext_iff]; exact d_is_sym

def i : matrix (fin 4) (fin 4) ℚ :=
![![0, 1, 0, 0],
  ![-1, 0, 0, 0],
  ![0, 0, 0, -1],
  ![0, 0, 1, 0]]

def j : matrix (fin 4) (fin 4) ℚ :=
![![0, 0, 1, 0],
  ![0, 0, 0, 1],
  ![-1, 0, 0, 0],
  ![0, -1, 0, 0]]

def k: matrix (fin 4) (fin 4) ℚ :=
![![0, 0, 0, 1],
  ![0, 0, -1, 0],
  ![0, 1, 0, 0],
  ![-1, 0, 0, 0]]

@[simp] lemma i_is_skewsym : iᵀ = - i := by dec_trivial
@[simp] lemma j_is_skewsym : jᵀ = - j := by dec_trivial
@[simp] lemma k_is_skewsym : kᵀ = - k := by dec_trivial

@[simp] lemma i_mul_i : (i · i) = -1 := by simp [i]; dec_trivial
@[simp] lemma j_mul_j : (j · j) = -1 := by simp [j]; dec_trivial
@[simp] lemma k_mul_k : (k · k) = -1 := by simp [k]; dec_trivial
@[simp] lemma i_mul_j : (i · j) = k := by simp [i, j, k]; dec_trivial
@[simp] lemma i_mul_k : (i · k) = -j := by simp [i, j, k]; dec_trivial
@[simp] lemma j_mul_i : (j · i) = -k := by simp [i, j, k]; dec_trivial
@[simp] lemma k_mul_i : (k · i) = j := by simp [i, j, k]; dec_trivial
@[simp] lemma j_mul_k : (j · k) = i := by simp [i, j, k]; dec_trivial
@[simp] lemma k_mul_j : (k · j) = -i := by simp [i, j, k]; dec_trivial
```

```
/-- `fin_23_shift` normalizes `λ (j : fin 23), f (s j)` in `![]` form,
    where `s : fin 23 → fin 23` is a function shifting indices. -/
lemma fin_23_shift (f : fin 23 → ℚ) (s : fin 23 → fin 23) :
(λ (j : fin 23), f (s j)) =
![f (s 0), f (s 1), f (s 2), f (s 3), f (s 4), f (s 5), f (s 6), f (s 7),
  f (s 8), f (s 9), f (s 10), f (s 11), f (s 12), f (s 13), f (s 14), f (s 15),
  f (s 16), f (s 17), f (s 18), f (s 19), f (s 20), f (s 21), f (s 22)] :=
by {ext i, fin_cases i, any_goals {simp},}


@[simp] lemma eq_aux₀:
dot_product (λ (j : fin 23), a (0 - j)) a +
dot_product (λ (j : fin 23), b (0 - j)) b +
dot_product (λ (j : fin 23), c (0 - j)) c +
dot_product (λ (j : fin 23), d (0 - j)) d = 92 :=
by {unfold a b c d, norm_num}

@[simp] lemma eq_aux₁:
dot_product (λ (j : fin 23), a (1 - j)) a +
dot_product (λ (j : fin 23), b (1 - j)) b +
dot_product (λ (j : fin 23), c (1 - j)) c +
dot_product (λ (j : fin 23), d (1 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₂:
dot_product (λ (j : fin 23), a (2 - j)) a +
dot_product (λ (j : fin 23), b (2 - j)) b +
dot_product (λ (j : fin 23), c (2 - j)) c +
dot_product (λ (j : fin 23), d (2 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₃:
dot_product (λ (j : fin 23), a (3 - j)) a +
dot_product (λ (j : fin 23), b (3 - j)) b +
dot_product (λ (j : fin 23), c (3 - j)) c +
dot_product (λ (j : fin 23), d (3 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₄:
dot_product (λ (j : fin 23), a (4 - j)) a +
dot_product (λ (j : fin 23), b (4 - j)) b +
dot_product (λ (j : fin 23), c (4 - j)) c +
dot_product (λ (j : fin 23), d (4 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₅:
dot_product (λ (j : fin 23), a (5 - j)) a +
dot_product (λ (j : fin 23), b (5 - j)) b +
dot_product (λ (j : fin 23), c (5 - j)) c +
dot_product (λ (j : fin 23), d (5 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₆:
dot_product (λ (j : fin 23), a (6 - j)) a +
dot_product (λ (j : fin 23), b (6 - j)) b +
dot_product (λ (j : fin 23), c (6 - j)) c +
dot_product (λ (j : fin 23), d (6 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}
```

```
@[simp] lemma eq_aux₇:
dot_product (λ (j : fin 23), a (7 - j)) a +
dot_product (λ (j : fin 23), b (7 - j)) b +
dot_product (λ (j : fin 23), c (7 - j)) c +
dot_product (λ (j : fin 23), d (7 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₈:
dot_product (λ (j : fin 23), a (8 - j)) a +
dot_product (λ (j : fin 23), b (8 - j)) b +
dot_product (λ (j : fin 23), c (8 - j)) c +
dot_product (λ (j : fin 23), d (8 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₉:
dot_product (λ (j : fin 23), a (9 - j)) a +
dot_product (λ (j : fin 23), b (9 - j)) b +
dot_product (λ (j : fin 23), c (9 - j)) c +
dot_product (λ (j : fin 23), d (9 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₀:
dot_product (λ (j : fin 23), a (10 - j)) a +
dot_product (λ (j : fin 23), b (10 - j)) b +
dot_product (λ (j : fin 23), c (10 - j)) c +
dot_product (λ (j : fin 23), d (10 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₁:
dot_product (λ (j : fin 23), a (11 - j)) a +
dot_product (λ (j : fin 23), b (11 - j)) b +
dot_product (λ (j : fin 23), c (11 - j)) c +
dot_product (λ (j : fin 23), d (11 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₂:
dot_product (λ (j : fin 23), a (12 - j)) a +
dot_product (λ (j : fin 23), b (12 - j)) b +
dot_product (λ (j : fin 23), c (12 - j)) c +
dot_product (λ (j : fin 23), d (12 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₃:
dot_product (λ (j : fin 23), a (13 - j)) a +
dot_product (λ (j : fin 23), b (13 - j)) b +
dot_product (λ (j : fin 23), c (13 - j)) c +
dot_product (λ (j : fin 23), d (13 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₄:
dot_product (λ (j : fin 23), a (14 - j)) a +
dot_product (λ (j : fin 23), b (14 - j)) b +
dot_product (λ (j : fin 23), c (14 - j)) c +
dot_product (λ (j : fin 23), d (14 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₅:
dot_product (λ (j : fin 23), a (15 - j)) a +
```

```
dot_product (λ (j : fin 23), b (15 - j)) b +
dot_product (λ (j : fin 23), c (15 - j)) c +
dot_product (λ (j : fin 23), d (15 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₆:
dot_product (λ (j : fin 23), a (16 - j)) a +
dot_product (λ (j : fin 23), b (16 - j)) b +
dot_product (λ (j : fin 23), c (16 - j)) c +
dot_product (λ (j : fin 23), d (16 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₇:
dot_product (λ (j : fin 23), a (17 - j)) a +
dot_product (λ (j : fin 23), b (17 - j)) b +
dot_product (λ (j : fin 23), c (17 - j)) c +
dot_product (λ (j : fin 23), d (17 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₈:
dot_product (λ (j : fin 23), a (18 - j)) a +
dot_product (λ (j : fin 23), b (18 - j)) b +
dot_product (λ (j : fin 23), c (18 - j)) c +
dot_product (λ (j : fin 23), d (18 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₁₉:
dot_product (λ (j : fin 23), a (19 - j)) a +
dot_product (λ (j : fin 23), b (19 - j)) b +
dot_product (λ (j : fin 23), c (19 - j)) c +
dot_product (λ (j : fin 23), d (19 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₂₀:
dot_product (λ (j : fin 23), a (20 - j)) a +
dot_product (λ (j : fin 23), b (20 - j)) b +
dot_product (λ (j : fin 23), c (20 - j)) c +
dot_product (λ (j : fin 23), d (20 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₂₁:
dot_product (λ (j : fin 23), a (21 - j)) a +
dot_product (λ (j : fin 23), b (21 - j)) b +
dot_product (λ (j : fin 23), c (21 - j)) c +
dot_product (λ (j : fin 23), d (21 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

@[simp] lemma eq_aux₂₂:
dot_product (λ (j : fin 23), a (22 - j)) a +
dot_product (λ (j : fin 23), b (22 - j)) b +
dot_product (λ (j : fin 23), c (22 - j)) c +
dot_product (λ (j : fin 23), d (22 - j)) d = 0 :=
by {simp only [fin_23_shift, a, b ,c ,d], norm_num}

lemma equality :
A · A + B · B + C · C + D · D = (92 : ℚ) • (1 : matrix (fin 23) (fin 23) ℚ) :=
begin
  -- the first `simp` transfers the equation to the form `cir .. = cir ..`
```

```
  simp [cir_mul, cir_add, one_eq_cir, smul_cir],
  -- we then show the two `cir`s consume equal arguments
  congr' 1,
  -- to show the two vectors are equal
  ext i,
  simp [mul_vec, cir],
  -- ask lean to inspect the 23 pairs entries one by one
  fin_cases i,
  exact eq_aux₀,
  exact eq_aux₁,
  exact eq_aux₂,
  exact eq_aux₃,
  exact eq_aux₄,
  exact eq_aux₅,
  exact eq_aux₆,
  exact eq_aux₇,
  exact eq_aux₈,
  exact eq_aux₉,
  exact eq_aux₁₀,
  exact eq_aux₁₁,
  exact eq_aux₁₂,
  exact eq_aux₁₃,
  exact eq_aux₁₄,
  exact eq_aux₁₅,
  exact eq_aux₁₆,
  exact eq_aux₁₇,
  exact eq_aux₁₈,
  exact eq_aux₁₉,
  exact eq_aux₂₀,
  exact eq_aux₂₁,
  exact eq_aux₂₂,
end
end H_92

open H_92

def H_92 := A ⊗ 1 + B ⊗ i + C ⊗ j + D ⊗ k

/-- Poves every entry of `H_92` is `1` or `-1`. -/
lemma H_92.one_or_neg_one : ∀ i j, (H_92 i j) = 1 ∨ (H_92 i j) = -1 :=
begin
  rintros ⟨c, a⟩ ⟨d, b⟩,
  simp [H_92, Kronecker],
  fin_cases a,
  any_goals {fin_cases b},
  any_goals {norm_num [one_apply, i, j, k]},
end

/-- Proves `H_92 · H_92ᵀ` is a diagonal matrix. -/
lemma H_92_mul_transpose_self_is_diagonal : (H_92 · H_92ᵀ).is_diagonal :=
begin
  simp [H_92, transpose_K, matrix.mul_add, matrix.add_mul, K_mul,
  cir_mul_comm _ a, cir_mul_comm c b, cir_mul_comm d b, cir_mul_comm d c],
  have :
  (cir a · cir a)⊗1 + -(cir a · cir b)⊗i + -(cir a · cir c)⊗j + -(cir a · cir d)⊗k +
  ((cir a · cir b)⊗i + (cir b · cir b)⊗1 + -(cir b · cir c)⊗k + (cir b · cir d)⊗j) +
  ((cir a · cir c)⊗j + (cir b · cir c)⊗k + (cir c · cir c)⊗1 + -(cir c · cir d)⊗i) +
  ((cir a · cir d)⊗k + -(cir b · cir d)⊗j + (cir c · cir d)⊗i + (cir d · cir d)⊗1) =
```

```
  (cir a · cir a)⊗1 + (cir b · cir b)⊗1 + (cir c · cir c)⊗1 + (cir d · cir d)⊗1 :=
  by abel,
  rw this, clear this,
  simp [←add_K, equality], -- uses `equality`
end

@[instance]
theorem Hadamard_matrix.H_92 : Hadamard_matrix H_92 :=
⟨H_92.one_or_neg_one,
 mul_tranpose_is_diagonal_iff_has_orthogonal_rows.1 H_92_mul_transpose_self_is_diagonal⟩

end order_92
/- ## end order 92-/

/- ## order -/
section order
open matrix Hadamard_matrix

theorem Hadamard_matrix.order_constraint
[decidable_eq I] (H : matrix I I ℚ) [Hadamard_matrix H]
: card I ≥ 3 → 4 ∣ card I :=
begin
  intros h, -- h: card I ≥ 3
  -- pick three distinct rows i₁, i₂, i₃
  obtain ⟨i₁, i₂, i₃, ⟨h₁₂, h₁₃, h₂₃⟩⟩:= pick_elements h,
  -- the cardinalities of J₁, J₂, J₃, J₄ are denoted as i, j, k, l in the proof in words
  set J₁ := {j : I | H i₁ j = H i₂ j ∧ H i₂ j = H i₃ j},
  set J₂ := {j : I | H i₁ j = H i₂ j ∧ H i₂ j ≠ H i₃ j},
  set J₃ := {j : I | H i₁ j ≠ H i₂ j ∧ H i₁ j = H i₃ j},
  set J₄ := {j : I | H i₁ j ≠ H i₂ j ∧ H i₂ j = H i₃ j},
  -- dₘₙ proves Jₘ Jₙ are disjoint
  have d₁₂: disjoint J₁ J₂,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros, linarith},
  have d₁₃: disjoint J₁ J₃,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d, exact c a},
  have d₁₄: disjoint J₁ J₄,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d, exact c a},
  have d₂₃: disjoint J₂ J₃,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d, exact c a},
  have d₂₄: disjoint J₂ J₄,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d, exact c a},
  have d₃₄: disjoint J₃ J₄,
  {simp [set.disjoint_iff_inter_eq_empty], ext, simp, intros a b c d,
  have : H i₁ x = H i₂ x, {linarith}, exact c this},
  -- u₁₂ proves J₁ ∪ J₂ = matched H i₁ i₂
  have u₁₂: J₁.union J₂ = matched H i₁ i₂,
  {ext, simp [J₁, J₂, matched, set.union], tauto},
  -- u₁₃ proves J₁ ∪ J₃ = matched H i₁ i₃
  have u₁₃: J₁.union J₃ = matched H i₁ i₃,
  {ext, simp [J₁, J₃, matched, set.union], by_cases g : H i₁ x = H i₂ x; simp [g]},
  -- u₁₄ proves J₁ ∪ J₄ = matched H i₂ i₃
  have u₁₄: J₁.union J₄ = matched H i₂ i₃,
  {ext, simp [J₁, J₄, matched, set.union], tauto},
  -- u₂₃ proves J₂ ∪ J₃ = mismatched H i₂ i₃
  have u₂₃: J₂.union J₃ = mismatched H i₂ i₃,
  { ext, simp [J₂, J₃, mismatched, set.union],
    by_cases g₁ : H i₂ x = H i₃ x; simp [g₁],
    by_cases g₂ : H i₁ x = H i₂ x; simp [g₁, g₂],
```

```
    exact entry_eq_entry_of (ne.symm g₂) g₁ },
  -- u₂₄ proves J₂ ∪ J₄ = mismatched H i₂ i₄
  have u₂₄: J₂.union J₄ = mismatched H i₁ i₃,
  { ext, simp [J₂, J₄, mismatched, set.union],
    by_cases g₁ : H i₁ x = H i₂ x; simp [g₁],
    split, {rintros g₂ g₃, exact g₁ (g₃.trans g₂.symm)},
    intros g₂,
    exact entry_eq_entry_of g₁ g₂ },
 -- u₃₄ proves J₃ ∪ J₄ = mismatched H i₁ i₂
  have u₃₄: J₃.union J₄ = mismatched H i₁ i₂,
  { ext, simp [J₃, J₄, matched, set.union],
    split; try {tauto},
    intros g₁,
    by_cases g₂ : H i₁ x = H i₃ x,
    { left, exact ⟨g₁, g₂⟩ },
    { right, exact ⟨g₁, entry_eq_entry_of g₁ g₂⟩ } },
  -- eq₁: |H.matched i₁ i₂| = |H.mismatched i₁ i₂|
  have eq₁ := card_match_eq_card_mismatch H h₁₂,
  -- eq₂: |H.matched i₁ i₃| = |H.mismatched i₁ i₃|
  have eq₂ := card_match_eq_card_mismatch H h₁₃,
  -- eq₃: |H.matched i₂ i₃| = |H.mismatched i₂ i₃|
  have eq₃ := card_match_eq_card_mismatch H h₂₃,
  -- eq : |I| = |H.matched i₁ i₂| + |H.mismatched i₁ i₂|
  have eq := card_match_add_card_mismatch H i₁ i₂,
  -- rewrite eq to |I| = |J₁| + |J₂| + |J₃| + |J₄|, and
  -- rewrite eq₁ to |J₁| + |J₂| = |J₃| + |J₄|
  rw [set.card_disjoint_union' d₁₂ u₁₂, set.card_disjoint_union' d₃₄ u₃₄] at eq₁ eq,
  -- rewrite eq₂ to |J₁| + |J₃| = |J₂| + |J₄|
  rw [set.card_disjoint_union' d₁₃ u₁₃, set.card_disjoint_union' d₂₄ u₂₄] at eq₂,
  -- rewrite eq₃ to |J₁| + |J₄| = |J₂| + |J₄|
  rw [set.card_disjoint_union' d₁₄ u₁₄, set.card_disjoint_union' d₂₃ u₂₃] at eq₃,
  -- g₂₁, g₃₁, g₄₁ prove that |J₁| = |J₂| = |J₃| = |J₄|
  have g₂₁ : J₂.card = J₁.card, {linarith},
  have g₃₁ : J₃.card = J₁.card, {linarith},
  have g₄₁ : J₄.card = J₁.card, {linarith},
  -- rewrite eq to |I| = |J₁| + |J₁| + |J₁| + |J₁|
  rw [g₂₁, g₃₁, g₄₁, set.univ_card_eq_fintype_card] at eq,
  use J₁.card,
  simp [eq], noncomm_ring,
end

theorem Hadamard_matrix.Hadamard_conjecture:
∀ k : ℕ, ∃ (I : Type*) [fintype I],
by exactI ∃ (H : matrix I I ℚ) [Hadamard_matrix H],
card I = 4 * k :=
sorry -- Here, `sorry` means if you ask me to prove this conjecture,
      -- then I have to apologize.

end order
/- ## end order -/

end Hadamard_matrix
/- ## end Hadamard_matrix  -/

end matrix
```