Computer Verification of Graph Theory and Algebraic Combinatorics Constructions

3rd year project report for candidate 1035522

Honour School of Computer Science - Part B

Submitted as part of an MCompSci in Computer Science

Trinity term, 2021

Abstract

The extensive use of combinatorics, linear algebra and analysis has undoubtedly increased the capabilities and efficiency of modern systems. Domains such as machine learning, computer graphics and artificial intelligence are based on rich mathematical backgrounds and operate with complex mathematical objects, such as regular graphs, elliptic curves and Hadamard matrices, by taking advantage of their mathematically-proved properties. Consequently, there is a considerable interest in having mathematical objects in computer-ready form, together with sound and complete proofs that would ensure the correctness of any system built upon them.

The project is centered around formalizing and contributing to the verification of two key areas in mathematics:

- **linear algebra**: documentation for eigenvalues and eigenvectors of real matrices [3].
- **spectral graph theory**: introduction to the incidence and Laplacian matrices of undirected simple graphs [4].

Acknowledgements

Many thanks to my supervisor Dmitrii Pasechnik for coming up with the idea of the project and for all the help and advice provided. Thanks also to Kevin Buzzard for teaching a very useful introductory course on the Lean proof assistant[1]. Finally, many thanks to the Lean community of developers from Zulip[2] for their prompt and quick guidance with the existing Lean main library *mathlib* and for the constructive feedback towards my contributions.

[1] `https://xenaproject.wordpress.com/category/imperial/formalising-mathematics-course/`
[2] `https://leanprover.zulipchat.com/#recent_topics`

Contents

[1] Whenever a theorem is stated, the mathematical proof will try to resemble the Lean code solution.

1. Introduction

1.1 Motivation

Commonly, the most reliable (or only available) form of proof for some key areas in mathematics is on paper. This motivates the need of computer checkable proofs via interactive theorem provers, one of the main areas of *formal verification*. The main advantage here is the re-usability of already established facts with the purpose of expanding towards more advanced concepts and theorems automatically.

In order to perform formal verification of the desired mathematical concepts and theorems, we require a reliable and easy-to-use proof assistant[1]. The choice for this project is *Lean*[1] as it has a clear syntax and a lot of resources available online in the form of *mathlib*[2] - the entire library of definitions and theorems. Lean can also be viewed as a functional language that is built upon type dependent theory, with automated tools for proofs. It mainly focuses on the "verification" part of theorem proving, where each claim is supported by a proof in a suitable (axiomatic) foundation and every step is justified by prior definitions and theorems, coming down to basic axioms.

1.2 Contributions

«««< HEAD The main goal of this project is to formally prove theorems about the incidence and Laplacian matrices of an undirected (loopless) graph. At the same time, the author noticed that the linear algebra area can be extended with properties of eigenvalues and eigenvectors of symmetric real matrices – and it is needed for the task at hand. The contributions consist of pull-requests to the main library *mathlib* and contain new definitions (which need to be consistent with the desired mathematical concepts) and new lemmas (which establish essential facts about the newly created objects). The contributions are based on existing documentation for undirected graphs and their adjacency matrices, vector and matrix operations, finite sets and unordered pairs. ======= The main goal of this project is to formally prove theorems about the incidence and Laplacian matrices of an undirected (loopless) graph. At the same time, the author noticed that the linear alge-

[1] `https://en.wikipedia.org/wiki/Proof_assistant`

bra area can be extended with properties of eigenvalues and eigenvectors of symmetric real matrices—and it is needed for the task at hand. The contributions consist of pull-requests to the main library *mathlib* and contain new definitions (which need to be consistent with the desired mathematical concepts) and new lemmas (which establish essential facts about the newly created objects). The contributions are based on existing documentation for undirected graphs and their adjacency matrices, vector and matrix operations, finite sets and unordered pairs. »»»> 874c91c37e7e9bf2ef23383d78d9e8eaade51755

1.3 Requirements

(A) Each structure (mathematical object) must be rigorously and correctly defined and each lemma or theorem[1] must be mathematically-justified.
(B) Where possible, each definition or theorem will be as general as possible with respect to types (the least restrictive ones) and assumptions (only non-redundant ones required).
(C) Each proof object will maximise the use of already existing documentation, as long as this maintains the clarity and rigour of the proof.
(D) Each proof will minimise the amount of work Lean must do on its own (it is preferred to explicitly specify how a certain goal can be reached, even when it is trivial[2]).

1.4 Challenges

The first parts of this project involved getting familiar with the syntax of Lean and with the mathematical concepts that are to be proved. This was done by consulting several sources with different approaches to theorems to be stated, because choosing the most appropriate one was crucial in creating a documentation that is compatible with *mathlib*. This compatibility implied the use of a considerable amount of imports and rewrite tactics (concept to be fully explained later). Lastly, converting proofs between types (e.g. real to complex and vice-versa) and explicitly stating obvious mathematical transformations (e.g. $a = b \to b = a$) might seem tedious, but will be necessary.

[1] In Lean, lemma and theorem are precisely the same thing, so we will use them interchangeably.
[2] We can tell Lean to solve a goal of the form $a = a$ on its own, but we prefer to specify the lemma or tactic that states this explicitly (i.e. *reflexivity*).

1.5 Structure of report

The rest of the report is structured as follows: an introduction to Lean and how it can be used to prove "toy" lemmas is presented in Section 2. A concrete example of how to build an undirected graph (together with essential lemmas) is described in Section 3. The in-depth contribution to the linear algebra module is presented in Section 4. The implementation of the incidence and Laplacian matrices is discussed in Sections 5 and 6, respectively. Finally, in Section 7, we present the conclusions, limitations and future directions for this project.

2. Background : The Lean theorem prover

2.1 About Lean

Formal verification is the act of proving or disproving the validity of a given statement according to a given specification. The Lean project[5][6] was launched in 2013 by Leonardo de Moura, who was working at the Microsoft Research Redmond group. It is currently an open-source project, allowing and encouraging people to use and extend its main library (*mathlib*). The main goal is to gradually increase the quantity and quality of mathematical proofs and concepts that can be expressed and furthermore automatically checked. Lean can be used both as a functional language (with syntax similar to *Haskell* and *OCaml*) and as an interactive theorem prover, the latter being the main use for the current project.

2.2 Dependent Type Theory

One way to represent the foundation of mathematics is via **set theory**, which relies on the concept that all mathematical objects can be expressed by **sets**. Lean uses a different approach, called **type theory**, where each expression has an associated **type**. For instance, 1 can be seen as a natural number (type *nat*) or as an integer (type *int*):

```
#check 1            --  (by default 1 is considered natural)
#check (1 : )       --  (we can type cast it to integers)
#check 1 + (1 : ) --  (first 1 cast to integer)
```

The first occurrence in the code of a new piece of syntax will be highlighted, and then its use will be explained (the explanation will also use the highlighting). When asking the system to provide specific information, we use commands that begin with #, here #check returns the type of a given expression.

One particular property of type theory that will be used extensively by Lean is that new types can be derived from others. For instance, we can derive the type of functions $\alpha \to \beta$ from the types α and β, or their Cartesian product type α x β.

```
constant x : nat -- creates a new variable with an assigned type ()
constant g : nat  nat -- similar to the way Haskell functions are defined
constant f : (nat  nat)  nat 𝔼 nat
#check g x        --
#check f g        --  𝔼
#check (f g).fst --  (fst/snd - natural way to access elements of a pair)
```

Extending the simple type theory showed so far, the **dependent type theory** of Lean treats each type (e.g. nat, bool..) as an object of type *Type*.

```
#check nat  -- Type
#check bool -- Type
```

But what is the type of *Type*? It is defined to be *Type 1* and the type of *Type 1* is *Type 2* and so on, meaning that *Type (n+1)* contains all objects of type *Type n*. Polymorphism appears from the fact that types can have as a parameter other types, for instance the type of lists of elements over some type α:

```
constant  : Type
#check list   -- Type u_1  Type u_1 (u_1 is a placeholder)
#check list  -- Type
#check prod   -- Type u_1  Type u_2  Type (max u_1 u_2)
```

A pair of two elements of type α and β, should be part of the universe that contains both the types of its constituent elements, that is why the maximal type is used.

There is also a familiar way to define functions, using the lambda construction (λ) :

```
#check λ x : , x + 5 --   , can omit :, Lean infers it from context
#reduce ( x : , x) a -- a (simplifies given expression via reduction)
```

This is how we can **define** a new mathematical object:

```
-- def function_name parameters : function_type := function_body
def compose (   : Type*)(g :   )(f :   )(x : ) :  := g (f x)
```

The syntax shows that for a definition we can specify the parameters it applies to (in this case 3 general types, 2 functions and a variable), the type of the object that the definition creates (γ) and the object itself or how to create it.

We can also create local definitions with the let..in construct:

```
let y := x + 3 in y * (y + 1) -- equivalent to (x + 3) * (x + 4)
```

Since the use of the *constant* keyword introduces new fixed objects in our environment and this can lead to name inconsistencies in our code, the variable syntax will be used instead, as it has the same effect except that the instantiated objects can be constrained to a particular section, rather than having a global effect:

```
section test -- name can be omitted
variable y :
#check y     --  (defined locally to this section)
end test     -- all sections must be closed before the end of file
#check y     -- unknown identifier 'y'
```

A similar behaviour happens in the case of definitions and namespaces, in the sense that when a definition (*inc*) is created inside a namespace (*test*), we can refer to the definition

in the namespace as *inc*, but outside we need to use *test.inc*, unless we re-open the namespace:

```
namespace test
def inc (x : ) := x + 1
end test
#check inc      -- unknown identifier 'inc'
#check test.inc --
open test
#check inc      --
```

Sections and namespaces can be nested inside each other, but must be closed in the reverse order in which they were created.

Now, consider a function *cons* that takes an element of type α and appends it to a list of elements of type α. Since this function should be polymorphic, we can expect *cons* α to have type $\alpha \to$ list $\alpha \to$ list α. But since the type of *cons* **depends on its input**, it will have a "dependent function type" of the form Π x : α, β x, which means that for each a : α, *cons a* is an element of β *a*.

```
constant cons : Π  : Type*,   list   list  -- Type* = fresh type
```

With this definition, whenever the *cons* function will be used, the type α that defines it must be specified. When we have a type that can be **inferred** by other (**explicit**) types, we surround it by {} (Lean will try to infer it via a process called elaboration):

```
constant cons :  { : Type*},   list   list  --  = implicit arg
#check @cons  --   list   list
```

Supplying a function with its implicit arguments requires a @ before its name.

2.3 Propositions and proofs

Moving towards proving mathematical statements, Lean introduces the concept of *Prop*, which is a special form of *Type* construct. Intuitively, from an object P of type *Prop*, we can create a new type P which can either be empty, in the case that P is a false statement, or can contain at least one element H_P : P, which is a witness (proof) of the fact that P is true. Furthermore, if there are two objects that are both of type P, then they are equivalent, since all proofs of the same proposition should be equivalent. Therefore, for every statement P that we want to assert, the goal will be to create an object of type P. From now on, our task will be to construct a proposition which will reflect the mathematical statement that we want to prove (P : *Prop*) and also to provide

Lean with an object H_P : P, which will serve as a proof that P holds.

2.4 Tactics

We will now expose the main tools for proving mathematical statements - tactics. Some of them will gradually be introduced here via toy examples, others will appear and will be explained as we provide the main part of the project code. Each command (tactic) will we followed by a comma (,) after which the command will be applied to the current goal.

```
section toy_examples
variable P : Prop -- type of basic propositions

theorem not_not : P  ň ň( P) :=
begin -- start of the proof
  intro H_P, -- H_P : P  ňňP
  have H_not :  Q : Prop, ň Q  (Q  false),
  {intro Q, refl}, -- H_P, H_not :  Q : Prop, ň Q  (Q  false),  ňňP
  rw H_not P, -- H_P, H_not  ň(P  false)
  rw H_not (P  false), -- H_P, H_not  (P  false)  false
  intro H_nP, -- H_P, H_not, H_nP : P  false  false
  apply H_nP, -- H_P, H_not, H_nP  P
  exact H_P   -- goal accomplished
end -- end of the proof
```

The sole purpose of a theorem in Lean is to create an object of a specific type. When constructing a theorem of type P, we provide a witness to the fact that P can be proved. The role of tactics is to instruct Lean on how to create such an object, starting from basic axioms. Entering tactic mode can be signalled with a begin..end block.
We started with a statement that (for a fixed proposition P), we have $P \to \neg (\neg P)$.

- intro h can be applied to a goal $P \to Q$ and its effect is to **introduce** h : P as a hypothesis to our environment and transform the goal to Q. Furthermore, if the goal is of the form $\forall x : \alpha$, f x, the command *intro a* will create $a : \alpha$ and the goal will become f a (intuitively, we prove that for all x, f x by choosing an arbitrary a and proving f a).

- have h : P will **create** a new subgoal of type P, and will add h : P to the current hypotheses after the subgoal is proved; generally, we prove the subgoal in a new proof block, delimited with {}.

- refl will close any goal of the form A = B or A ↔ B, where A and B are **exactly the same thing**, in our case it will close the goal ¬ Q ↔ (Q → false) because this is how ¬ is defined by Lean.

- rw h, where h : A = B (or A ↔ B) will **rewrite** all As in the goal to Bs, checking if the resulting goal can be solved with the *refl* tactic, and if so, closing it. To change all Bs into As we write rw ← h.

- apply h, where h is a proof of P → Q will transform any goal of the form Q into the goal P (this is a form of backward reasoning: if we can prove P, then with the proof of P → Q we can prove Q).

- exact h will close any goal that only consists of h.

If we notice that the statement H_not could be re-used multiple times across multiple theorems, we can create a new lemma[1] to prove H_not.

```
lemma not_def {Q : Prop} : ň Q  (Q  false) := by refl

theorem not_not : P  ň ň( P) := begin
  intro H_P, -- H_P : P  ňňP
  repeat { rw not_def }, -- H_P  (P  false)  false
  simp [H_P] -- solves the goal using H_P
end
```

- by h is equivalent to *begin h end* and will be used for short proofs.

- repeat {<commands>} will **repeatedly** apply the entire block of commands to the goal until it solves it, or until the block no longer makes progress (notice that Q in *not_def* is substituted once for ¬ P and once for P, depending on what Lean infers).

- simp [h1, h2, ...] triggers an automatic procedure which will be applied to the current goal, in which Lean uses all the already existing lemmas with the @[simp] attribute plus the hypotheses (or definitions) h1, h2, ... to **simplify** the current goal as much as possible or even close it (infinite loops are avoided). We will see that the best practice is to explicitly tell Lean which are the tools that it can use to solve the current goal. This way, when a helping lemma used by the simplifier is changed, any uses of it can be easily tracked.

[1] As a matter of style, the main statements will be **theorems** and the required helping proofs will be **lemmas**.

```
theorem not_not : ň ň( P)  P := begin
  intro H_nnP, -- H_nnP: ňňP  P
  by_cases H : P, -- create two subgoals depending on the truth value of
    P
  -- H : P
  { assumption },
  -- H : ň P
  { exfalso, -- we know that H and H_nnP contradict each other, so
    together they imply false
    apply H_nnP H } -- ňň P considered as ň P  false, by definition
end
```

- by_cases h : P will do a case analysis on whether the proposition P is true or false.
- assumption looks through the hypotheses in the context of the current goal, and if there is one matching the conclusion, it closes the goal.
- exfalso transforms the goal to false (useful when hypotheses lead to a contradiction).

```
theorem not_not : ň ň( P)  P := begin
  intro hyp, -- hyp: ňňP
  by_contradiction h, -- h: ňP
  exact hyp h -- use that ňP is equivalent to P  false, by definition
end
```

- by_contradiction h will transform any goal P into false and will add h : ¬ P to the set of current assumptions (equivalent to proof by contradiction).

```
lemma union_self (X : set ) : X  X = X := begin
  ext a, -- a  X  X  a  X
  split, -- a  X  X  a  X / a  X  a  X  X
  { intro h, -- h: a  X  X, equivalent to a  X  a  X
    cases h with ha ha; -- ha: a  X
    exact ha }, -- solve both goals with the command (;)
  { intro ha, -- ha : a  X, goal is a  X  a  X
    left, -- choose to prove the left subgoal (both goals are the same)
    exact ha }
end
```

- ext a will transform a goal of the form A = B for two sets (or other "extensionable" objects signalled with @[ext] as we will see in the next section) into showing that a ∈ A ↔ a ∈ B.
- split will transform a goal of the form P ∧ Q into two new goals : P and Q; P ↔ Q is definitionally equivalent to P → Q ∧ Q → P.
- the behaviour of cases h depends on the form of h:

  - h : P ∧ Q - cases h with hP hQ creates two new hypotheses hP : P and hQ : Q
  - h : P ∨ Q - cases h with hP hQ will create two instances of the problem, one where we have hP : P instead of h, and one where we have hQ : Q. We can apply a command H to all the current goals by using ; to the previous command (*<command>; H*)

- left (and the corresponding right) will transform a goal of the form P ∨ Q to P (respectively Q); we only need to prove one of them for the disjunction to hold.

```
theorem anything_by_sorry : 1  1 := sorry
end toy_examples
```

- sorry magically solves every theorem, because its type matches any proposition, and it is used as an indication that the proof of the current statement is missing. Statements that are proved with *sorry* can be used by other statements, but the compiler will raise warning messages to alert that there is still work to be done.

2.5 Coercions

Throughout this project, we will work a lot with types that are restrictions on bigger types (e.g. the type of natural numbers ℕ is a restriction on the type of integers ℤ).

```
constants (n : nat) (z : int)
#check n + z -- "type mismatch" - z expected to have type
#check n + z -- , we specify that n is treated as integer with
```

We sometimes need to be explicit because Lean has restrictions on some of the operators it uses (here the addition operation only accepts arguments with the same type). When we have two types T and T' and we want to **coerce** one element from T to T', we need to provide an instance has_coe from T to T', to show Lean how to do the transformation.

```
instance : has_coe   :=  r, r, 0
constants (r : ) (c : )
#check r + c --
```

For instance, to coerce a real element r into a complex one, simply create a complex object with real value r and complex value 0.

3. Practical example : Undirected graphs

In order to get familiarised even more with how we will work in the next sections, we will look into a more concrete example of an **already implemented** library in Lean, which will help us build our own mathematical objects in the next sections.

3.1 Simple graph

An **undirected graph** can be defined as a pair (V, E), where V is a set of objects called **vertices** and E is a symmetric relation on V. The Lean library only considers undirected graphs with no self-loops i.e. no edge between any vertex and itself, and calls this construct a *simple graph*. The structure that defines the simple graphs is based on 4 fields:

- *V* - the type used for vertices.
- *adj* - a function that takes two elements of V and returns whether there is an edge between them or not (relation on V).
- *sym* - a proof that the relation adj is symmetric (edges are bidirectional).
- *loopless* - a proof that the relation adj is irreflexive (no self-loops in the graph).

```
@[ext]
structure simple_graph (V : Type u) :=
(adj : V  V  Prop)
(sym : symmetric adj) -- symmetric :  {x y}, x  y  y  x
(loopless : irreflexive adj) -- irreflexive :  x, ¬ x  x
```

- structure constructs a non-recursive inductive type S, together with *projection functions* to destruct each instance of S and retrieve the values that are stored in its *fields*. Its syntax follows *structure <name> <parameters> : <type> := <fields>*.
- @[ext] is helpful when showing that two structures are equivalent by proving that their corresponding fields are equivalent; the tactic *ext* will transform a goal G = G', where G and G' are two *simple_graphs* into G.adj v w ↔ G'.adj v w (introducing two fresh variables v, w into the context).

The *symmetric* and *irreflexive* properties are already defined in Lean, so there is no need to redefine them. There should be a way of constructing a graph from a given relation:

```
def simple_graph.from_rel{V : Type u}(r : V  V  Prop):simple_graph V :=
{ adj :=  a b, (a  b)  (r a b  r b a),
  sym :=  a b H_adj, ⟨H\_adj.1.symm, H\_adj.2.symm⟩,
  loopless :=  a hne, _, hne rfl}
```

- .symm applied to any equation *a op b* (with symmetric relation *op*) returns *b op a*.
- *adj* - starting from relation r and two elements, an edge is "created" iff the two elements are distinct and they are in any way related by r.
- *sym* - the symmetric property of *adj* must be established: for any two elements a and b and proof H_adj of their adjacency (a ≠ b ∧ (r a b ∨ r b a)), the fact that b and a are also adjacent follows from the symmetric property *.symm* of the ≠ and ∨ operators (if h : P ∧ Q, we can access the proof of P via h.1 and of Q via h.2, or we can **pattern match** h with ⟨hP, hQ⟩, where hP : P and hQ : Q).
- *loopless* - the irreflexive property of *adj* must be established: for any vertex a and proof H_adj of *adj* (pattern-matched keeping only the first part hne, which states that a ≠ a, using an _ to say that we do not care about the rest), the proof is exactly *hne rfl* (rfl is a proof that any object is equal to itself), which equals *false* (remember ¬ x ≺ x is equivalent to x ≺ x → *false*, and *false* is the goal here).

3.2 Complete graph - Inhabited

Complete graph - For any type of vertices V, we can create a (unique) undirected graph that has an edge between any two vertices, which is called the **complete graph** of the set defined by V. Since it is an instance of a simple graph (no self-loops), we can define it:

```
def complete_graph (V : Type u) : simple_graph V :=
{ adj := ne, sym :=  a b H_adj, H_adj.symm,
  loopless :=  a H_adj, H_adj rfl }
```

Two vertices are adjacent in the complete graph iff they are distinct (ne a b ↔ a ≠ b), and using the irreflexivity and symmetry of *ne*, the definition is complete.

Inhabited - Every type seen so far (nat, bool, list α etc.) has an underlying set of elements. To specify that a type T corresponds to a **non-empty** set of elements, an instance object inhabited will be used to attest that there is at least one element of type

T. The *simple_graph* structure is itself a polymorphic type over V with relation adj, so a good candidate to show that the type is non-empty is the *complete_graph* object:

```
instance (V : Type u) : inhabited (simple_graph V) := complete_graph V
```

Knowing that at least one *simple_graph* exists, a simple graph G can be fixed for future definitions and lemmas, all packed in a *namespace*. A simple (and very re-usable) example lemma is that any two adjacent vertices are distinct in G:

```
namespace simple_graph -- lemmas for a given graph G, re-use with G.lemma
variables {V : Type u} (G : simple_graph V)

lemma ne_of_adj {a b : V} (H_adj : G.adj a b) : a  b := begin
  intro hne, -- suppose a = b, the goal becomes "false"
  rw  hne at H_adj, -- H_adj becomes G.adj a a
  exact G.loopless a H_adj, -- implies false
end
```

3.3 Neighbour set

Sets S over a type T are defined as functions T $\to$ *Prop*, with (a : T) $\in$ S $\leftrightarrow$ S a. For such a function f, we can extract the elements a for which f a holds using the function set_of[1]. Therefore, the set of common neighbours of two vertices can be expressed as the intersection of the two neighbouring sets:

```
-- creates the set of vertices that are adjacent to vertex v in G
def neighbor_set (v : V) : set V := set_of (G.adj v)

def common_neighbors(v w : V):set V := G.neighbor_set v  G.neighbor_set w
```

3.4 Edge set - sym2

So far we have seen the product type $\alpha \times \beta$, but this defines **ordered pairs**. To emphasize that edges have no orientation, Lean has a type *sym2* α of unordered pairs for elements of type α. One can build a set of *sym2* α elements from a symmetric relation using the sym2.from_rel function (can get rid of prefix if we open the *sym2* module).

```
def edge_set : set (sym2 V) := sym2.from_rel G.sym

@[simp] -- simplifier will be encouraged to transform LHS in RHS
lemma mem_edge_set {v w : V} : ⟦(v, w)⟧  G.edge_set  G.adj v w :=
by refl -- lemma follows immediately from the definition of edge_set
```

An **unordered pair** is represented with double squared brackets ⟦(x, y)⟧.

[1] The use of this new color will become clear at the end of the section.

3.5 Incidence set

Having defined the edge set of the graph, the **incidence set** of a vertex v will only consist of the edges that are incident to v.

```
def incidence_set (v : V) : set (sym2 V) := {e  G.edge_set | v  e}
```

Clearly, the incidence set of any vertex v is included in the edge set of the graph:

```
lemma incidence_set_subset (v : V) : G.incidence_set v  G.edge_set :=
begin
  -- when applying intro x to goal A  B, goal becomes x  A  x  B
  intros e h_e,                 -- h_e: e  G.incidence_set v
  dsimp [incidence_set] at h_e, -- h_e: e  G.edge_set  v  e
  exact h_e.1
end
```

- dsimp [def1, def2, ...] uses the same mechanism as *simp*, but focuses on only using definitions for simplification, in this case using *incidence_set*.

We can say that an element a : α belongs to an unordered pair e : sym2 α with a $\in$ e. The **other** element from the pair can be obtained using the function other'.

```
def other_vertex_of_incident {v : V} {e : sym2 V}
  (h : e  G.incidence_set v) : V := h.2.other'
```

3.6 Fintypes & finsets

In the next sections, we will define matrices indexed by the vertices or edges of a simple graph, therefore it will be necessary to impose the constraint that the types and their underlying set are **finite**. With *fintype* α we impose that the underlying set of α has finitely many elements. The corresponding set will have type *finset* α.

```
def edge_finset [decidable_eq V] [fintype V] [decidable_rel G.adj] :
  finset (sym2 V) := set.to_finset G.edge_set -- to_finset : set  finset
```

To transform a set of elements into a *finset*, we must make sure that the corresponding type is finite (*fintype*). The extra requirements that any two elements of V can be compared with equality (*decidable_eq V*), that V is finite (*fintype V*) and that the adjacency relation is decidable (*decidable_rel G.adj*) are sufficient to show that the type $\uparrow$*G.edge_set* is finite.

```
variables (v : V) [fintype (G.neighbor_set v)]
def neighbor_finset : finset V :=
  (G.neighbor_set v).to_finset

def incidence_finset [decidable_eq V] : finset (sym2 V) :=
  (G.incidence_set v).to_finset
```

The ↑ operator is another form of **coercion**, in our case one that transforms a set *G.edge_set* of elements of type *sym2 V* into the type ↑*G.edge_set*, which only contains the edges that appear in the graph (↑*G.edge_set* is a **subtype** of *sym2 V*).

3.7 Degree

Since we are under the assumption that each neighboring set of any vertex v is finite, we can define the **degree** of v as the cardinality of its finite set of neighbors:

```
def degree :  := (G.neighbor_finset v).card
```

3.8 Adjacency matrix

One last important notion that we will intensively use is the **adjacency matrix**[1] of a simple graph G. It will be indexed on the vertex set V of the graph, and the corresponding value of an entry (i, j) will be 1 iff i and j are adjacent in G and 0 otherwise. Since its entries will be 0 or 1, it will be defined on a *semiring*[2] *R* for maximum generality.

```
universes u v -- they are used to distinguish types build from them
variables {V : Type u} [fintype V]
variables (R : Type v) [semiring R]

namespace simple_graph
variables (G : simple_graph V) (R) [decidable_rel G.adj]

def adj_matrix : matrix V V R
| i j := if (G.adj i j) then 1 else 0
```

Together with each of the definitions above, there are a lot of already proven lemmas, but there is no point in explicitly stating them or their proofs unless they will be useful for what we want to achieve in the following sections. Whenever we first encounter such a lemma, we will highlight it and then we will state what it achieves. Lemmas that were needed, but were absent from *mathlib* will be underlined. Some of this lemmas will have some additional requirements (e.g the type α is a monoid or ring), but unless this information is crucial, these requirements will be intentionally omitted for simplicity (Lean will still do the checks behind the scenes, as they are crucial for the correctness of our solutions).

[1] https://en.wikipedia.org/wiki/Adjacency_matrix#
[2] A ring without the requirement that each element must have an additive inverse.

4. Eigenvalues & eigenvectors of matrices

In order to make progress towards working with the eigenvalues of the adjacency or Laplacian matrices of undirected graphs, we must first define what an eigenvalue represents and then show some fundamental properties that can be generated from this concept. An inventory of the theory behind what will be proven in this section can be found at [7].

4.1 Main definitions

Let M be an n × n matrix with real entries. An **eigenvector** of M is a complex-valued non-zero vector **x** such that M **x** = μ **x**, for some complex value μ. μ is called the **eigenvalue** of M with corresponding eigenvector **x** (in Lean the multiplication between a scalar and a vector or matrix will be represented with $\mu \bullet$ **x** and it is called *smul*). We can already notice a small inconvenience that will appear very frequently in our work: we defined M to be real-valued and **x** to be complex-valued. On paper, this multiplication causes no problems because we know that all real values are also complex, however in Lean we must be very careful with this. The solution is to use **coercion** to define the product M **x** as the one between M coerced to a complex-valued matrix and **x**. This coercion will be made element-wise and shown in Section 2.5:

```
instance : has_coe (n  ) (n  ) :=  x, ( i, x i, 0)
instance : has_coe (matrix m n ) (matrix m n ) :=
   M, ( i j, M i j, 0)
```

In order to create a coercion between types A and B, we need to provide a function of type A → B which will describe how the transformation will happen: in the case of the matrix, for each real-valued M, return a matrix which when asked for entry (i, j) will provide a complex object with real part M i j and imaginary part 0 ⟨M i j, 0⟩. Now that M has the correct type, we will require the already defined vector-matrix multiplication operators:

```
def dot_product [has_mul ] [add_comm_monoid ] (v w : m  ) :  :=
 i, v i * w i

def mul_vec [semiring ] (M : matrix m n ) (v : n  ) : m
| i := dot_product ( j, M i j) v -- M  v

def vec_mul [semiring ] (v : m  ) (M : matrix m n ) : n
| j := dot_product v ( i, M i j) -- v  M
```

They are defined for a general type α, for which the minimum amount of requirements are specified between squared brackets (here $\mathbb{R}$ and $\mathbb{C}$ have the necessary properties). Now we are ready to define the eigenvalues and eigenvectors of a real-valued square matrix:

```
namespace matrix

-- coercion process described above
def Coe (M : matrix m n ) := (M : matrix m n )

variables (M : matrix n n )

def has_eigenpair ( : ) (x : n  ) : Prop :=
  x  0  (mul_vec M.Coe x =  x)

def has_eigenvector (x : n  ) : Prop :=
    : , M.has_eigenpair  x

def has_eigenvalue ( : ) : Prop :=
   x : n  , M.has_eigenpair  x
```

4.2 General lemmas[1]

Lemma 4.1 *Let M be a real $n \times n$ matrix, $\boldsymbol{x}$ be one of its eigenvectors, and $a \in \mathbb{C}^*$. Then $a\boldsymbol{x}$ is also an eigenvector of M.*

Proof. Since $\mathbf{x}$ is an eigenvector of M, there must exist an eigenvalue μ with M $\mathbf{x} = \mu$ $\mathbf{x}$ and also $\mathbf{x} \neq \mathbf{0}$. Then, it can be proved that a$\mathbf{x}$ is also an eigenvector of M using the same eigenvalue μ: first, suppose by contradiction that a$\mathbf{x} = \mathbf{0}$. Then a $= 0$ or $\mathbf{x} = \mathbf{0}$, both cases reaching a contradiction. Secondly, M (a$\mathbf{x}$) = a (M $\mathbf{x}$) = a (μ $\mathbf{x}$) = μ (a$\mathbf{x}$), which concludes the proof.

```
theorem has_eigenvector_smul (a : ) (x : n  ) (H_na : a  0)
   (H_eigenvector : has_eigenvector M x) : has_eigenvector M (a  x) :=
begin
  rcases H_eigenvector with , H_nx, H_mul ,
  use , -- corresponding eigenvalue
  split,
  { intro hyp, rw smul_eq_zero at hyp, tauto }, -- a  x  0
  calc (M.Coe).mul_vec (a  x)
      = a  M.Coe.mul_vec x : -- M  (a  x) = a  (M  x)
  by { rw mul_vec_smul_assoc }
  ... = a  (  x) :                    -- ... = a  (  x)
  by { rw H_mul }
  ... =   (a  x) :                    -- ... =   (a  x)
  by { simp only [smul_smul, mul_comm] }
end
```

[1] Whenever a theorem is stated, the mathematical proof will try to resemble the Lean code solution.

- rcases h is the recursive option of *cases*, in which the hypothesis h is structurally decomposed in pieces e.g if h is of the form ∃ x, P ∧ Q, *rcases h with* ⟨*x,* ⟨*hP, hQ*⟩ ⟩ creates a variable x and two new hypotheses hP : P and hQ : Q.

- use h instantiates the first term of a goal e.g if the goal is ∃ x, P(x), *use* μ will transform it into P(μ).

- tauto decomposes all the current hypotheses into their most simple components, applying them in all possible combinations to the goal until in closes it.

- calc uses the transitivity of equality to prove a goal of the form a = b via subgoals $a = a_1$, $a_1 = a_2$, ..., $a_{n-1} = a_n$, $a_n = b$ (each subgoal will be proved separately).

- simp only [h1, h2,...] simplifies the goal using **only** the hypotheses h1, h2,...; it is the general standard of *mathlib* to use this version of *simp* because this way can be seen which lemmas are actually useful to proving the goal and when modifications to some of the lemmas are made, it will be easy to track the further changes required.

```
lemma smul_eq_zero {c : } {x : } : c  x = 0  c = 0  x = 0
lemma mul_vec_smul_assoc (A : matrix m n ) (b : n  ) (a : ) :
  A.mul_vec (a  b) = a  (A.mul_vec b)
lemma smul_smul (a a : ) (b : ) : a  a  b = (a * a)  b
lemma mul_comm :  a b : , a * b = b * a
```

Lemma 4.2 *Let M be a real n × n matrix and two eigenvectors* $\boldsymbol{v}$ *and* $\boldsymbol{w}$ *with the same eigenvalue* μ*. Then, any* ***non-zero*** *linear combination* $a\boldsymbol{v} + b\boldsymbol{w}$ *is also an eigenvector of M with corresponding eigenvalue* μ*.*

Proof. First, we need to show that $a\mathbf{v} + b\mathbf{w} \neq \mathbf{0}$. This comes clearly from the lemma statement[1]. Secondly, we have M ($a\mathbf{v} + b\mathbf{w}$) = M ($a\mathbf{v}$) + M ($a\mathbf{w}$) = a (M $\mathbf{v}$) + a (M $\mathbf{w}$) = a (μ $\mathbf{v}$) + a (μ $\mathbf{w}$) = μ ($a\mathbf{v} + b\mathbf{w}$), which concludes the proof.

```
theorem has_eigenpair_linear (a b : ) (v w : n  ) ( : ) (H_ne : a  v + b
    w  0) (H : has_eigenpair M  v) (H : has_eigenpair M  w) :
   has_eigenpair M  (a  v + b  w) :=
begin
  rcases H with H, H,
  rcases H with H, H,
  use H_ne, -- a  v + b  w  0
```

[1] This specification is missing from a lot of mathematical sources, Lean however never leaves room for omissions like this, it cannot close a goal when the set of hypotheses is incomplete.

```
  calc M.Coe.mul_vec (a  v + b  w)
      = M.Coe.mul_vec(a  v) + M.Coe.mul_vec(b  w) :
  by { ext ; simp only [mul_vec, pi.add_apply, dot_product_add] }
  ... = a  M.Coe.mul_vec v + b  M.Coe.mul_vec w :
  by { ext ; simp only [mul_vec, algebra.id.smul_eq_mul,
                        dot_product_smul, pi.add_apply, pi.smul_apply] }
  ... = a  (  v) + b  (  w) :
  by { rw [H, H] }
  ... =   (a  v + b  w) :
  by { simp only [smul_smul, mul_comm, smul_add] }
end
```

We will often use the commands ext ; together when providing a proof that two vectors or matrices with complex entries are equal. The goal (after *ext*) becomes an equality between complex numbers, which requires the equality of their real and imaginary parts, so two subgoals will be created, and by using ; the next command will be applied to both goals.

```
lemma add_apply : (x + y) i = x i + y i -- we care about vectors here
lemma dot_product_add (u v w : m  ) :
  dot_product u (v + w) = dot_product u v + dot_product u w
lemma smul_eq_mul (x y : ) : x  y = x * y
lemma dot_product_smul (x : ) (v w : m  ) :
  dot_product v (x  w) = x * dot_product v w
lemma smul_apply (s : ) : (s  x) i = s  x i
lemma smul_add (a : M) (b b : A) : a  (b + b) = a  b + a  b
```

4.3 Symmetric matrices

First, let us introduce the concept of complex conjugate applied to complex-valued vectors (the complex conjugate of complex numbers conj is already defined in *mathlib*):

```
def vec_conj (x : n  ) : n  := i : n, conj (x i)
```

We will now define the concept of **symmetric matrix** M in Lean and prove theorems about its eigenvalues and eigenvectors:

Lemma 4.3 *Let M be a real symmetric $n \times n$ matrix with eigenvalue $\mu \in \mathbb{C}$ and corresponding complex-valued eigenvector $\boldsymbol{x}$. Then $\mu \in \mathbb{R}$.*

Proof.[1] By assumption, (1) M $\mathbf{x} = \mu\ \mathbf{x}$. Taking the complex conjugate of both sides, (M $\mathbf{x}$)* = (μ $\mathbf{x}$)* i.e (2) M $\mathbf{x}^* = \mu^*\ \mathbf{x}^*$. It can shown that (3) $\mu(\mathbf{x}^*)^\intercal\ \mathbf{x} = \mu^*(\mathbf{x}^*)^\intercal\ \mathbf{x}$ (details in the Lean proof below). Thus, (4) (μ - μ^*) $(\mathbf{x}^*)^\intercal\ \mathbf{x} = \mathbf{0}$. Using $(\mathbf{x}^*)^\intercal\ \mathbf{x} = \mathbf{0}$

[1] https://www.doc.ic.ac.uk/~ae/papers/lecture05.pdf

iff $\mathbf{x} = \mathbf{0}$ ($(\mathbf{x}^*)^\intercal\ \mathbf{x}$ = sum of norms of elements, which are all non-negative) and the fact that $\mathbf{x} \neq \mathbf{0}$ (as eigenvector of M), we get μ - $\mu^* = 0 \leftrightarrow \mu = \mu^* \leftrightarrow \mu \in \mathbb{R}$.

```
def symm_matrix : Prop := M = M

theorem symm_matrix_real_eigenvalues (H_symm : symm_matrix M) :
   ( : ), has_eigenvalue M   .im = 0 := begin
  -- (1) M x =  x
  rintro  x, H_x, H_eq,
  -- (2) M x* = * x*
  have H_eq : mul_vec M.Coe (vec_conj x) = (conj )  (vec_conj x),
  { rw [ vec_conj_smul  x,  M.vec_conj_mul_vec_re x, H_eq] },
  -- (3)  ((x*) x) = * ((x*) x)
  have H_eq :  * (dot_product (vec_conj x) x) =
               conj  * dot_product (vec_conj x) x,

  calc  * dot_product (vec_conj x) x
     = dot_product (vec_conj x) (  x) :   --  ((x*) x) = (x*) ( x)
  by { rw  smul_dot_product  (vec_conj x) x,
       simp [dot_product, vec_conj,  mul_assoc, mul_comm] }
  ... = dot_product (vec_conj x) (M.Coe.mul_vec x) :  -- ... = (x*) (M x)
  by { rw  H_eq }
  ... = dot_product (M.Coe.vec_mul (vec_conj x)) x :  -- ... = ((x*) M) x
  by { exact (dot_product_assoc (vec_conj x) M.Coe x).symm }
  ... = dot_product (M.Coe.mul_vec (vec_conj x)) x :-- ... = (M x*) x
  by { rw  mul_vec_transpose M.Coe (vec_conj x) }
  ... = dot_product (M.Coe.mul_vec (vec_conj x)) x :  -- ... = (M x*) x
  by { have H : M.Coe = M.Coe, { unfold Coe, tidy }, rw  H }
  ... = dot_product (conj   vec_conj x) x :          -- ... = (* x*) x
  by { rw H_eq }
  ... = conj  * dot_product (vec_conj x) x :          -- ... = * ((x*) x)
  by { rw  smul_dot_product (conj ) (vec_conj x) x },

  -- (4) ( - *) ((x*) x) = 0
  have H_eq : ( - conj ) * dot_product (vec_conj x) x = 0,
  { rw sub_mul, simp only [H_eq, sub_self] },
  --  - * = 0  (x*) x = 0
  rw mul_eq_zero at H_eq,
  cases H_eq with H_ H_prod,
  { rw [sub_eq_zero, eq_comm, eq_conj_iff_real] at H_,
    cases H_ with r H_r,
    rw [H_r, of_real_im] }, --  - * = 0
  { exfalso,
    exact H_x (vec_norm_sq_zero H_prod) } -- (x*) x = 0
end
```

- rintro is a combination of the *intro* and *rcases* tactics which allows for destructuring patterns while also introducing new variables to the environment.
- unfold iteratively replaces all occurrences of defs in the goal with their equations.
- tidy works in a similar manner to *simp*, but it focuses more on definitional equality

and has a different set of tactics which it repeatedly tries to apply to the goal (always used for closing goals, not for simplifying them).

```
lemma vec_conj_smul ( : ) (x : n  ) :
  vec_conj (  x) = (conj )  (vec_conj x)
lemma vec_conj_mul_vec_re (x : n  ) :
  vec_conj (mul_vec M.Coe x) = mul_vec (M.Coe) (vec_conj x)
lemma smul_dot_product (x : ) (v w : m  ) :
  dot_product (x  v) w = x * dot_product v w
lemma mul_assoc :  a b c : G, a * b * c = a * (b * c)
lemma dot_product_assoc (u : m  ) (v : m  n  ) (w : n  ) :
  dot_product ( j, dot_product u ( i, v i j)) w =
  dot_product u ( i, dot_product (v i) w)
lemma mul_vec_transpose (A : matrix m n ) (x : m  ) :
  mul_vec A x = vec_mul x A
lemma sub_mul (a b c : ) : (a - b) * c = a * c - b * c
lemma sub_self (a : ) : a - a = 0
lemma sub_eq_zero (a b : ) : a - b = 0  a = b
lemma eq_comm {a b : } : a = b  b = a
lemma eq_conj_iff_real {z : } : conj z = z   r : , z = r
lemma of_real_im (r : ) : (r : ).im = 0
lemma vec_norm_sq_zero {x : n  }
  (H_dot : dot_product (vec_conj x) x = 0) : x = 0
```

We should now introduce the notions of real and complex parts of complex-valued vectors:

```
def vec_re (x : n  ) : n   :=  i : n, (x i).re
def vec_im (x : n  ) : n   :=  i : n, (x i).im
```

Lemma 4.4 *For every real eigenvalue of a symmetric matrix M, there exists a corresponding real-valued eigenvector.*[1]

Proof. Let μ be an arbitrary eigenvalue of M and $\mathbf{x}$ its corresponding eigenvector. From Lemma 4.3 we know that $\mu \in \mathbb{R}$. We will distinguish two cases:

1) $\mathrm{Re}(\mathbf{x}) = \mathbf{0} \Rightarrow \mathrm{i}\mathbf{x} \in \mathbb{R}$. We argue that it is also an eigenvector of M. First, $\mathrm{i}\mathbf{x} = \mathbf{0}$ implies that $\mathbf{x} = \mathbf{0}$, which contradicts $\mathbf{x}$ being an eigenvector of M. Secondly, M (i$\mathbf{x}$) = i (M $\mathbf{x}$) = i (μ $\mathbf{x}$) = μ (i$\mathbf{x}$). Therefore, i$\mathbf{x}$ is a real-valued eigenvector of M with eigenvalue μ.

2) $\mathrm{Re}(\mathbf{x}) \neq \mathbf{0} \Rightarrow$ We argue that $\mathbf{x} + \mathbf{x}^*$ is a real-valued eigenvector of M with corresponding eigenvalue μ. First, $\mathbf{x} + \mathbf{x}^* = 2\ \mathrm{Re}(\mathbf{x}) \in \mathbb{R}$. Secondly, $\mathrm{Re}(\mathbf{x}) \neq \mathbf{0} \Rightarrow \mathbf{x} + \mathbf{x}^* \neq \mathbf{0}$[2]. Finally, M ($\mathbf{x} + \mathbf{x}^*$) = μ ($\mathbf{x} + \mathbf{x}^*$) (details in the Lean proof) concluding the proof.

```
theorem symm_matrix_real_eigenvectors (H_symm : symm_matrix M) ( : )
  (H_eigenvalue : has_eigenvalue M ) :
```

[1] https://sharmaeklavya2.github.io/theoremdep/nodes/linear-algebra/eigenvectors/real-matrix-with-real-eigenvalue-has-real-eigenvectors.html

[2] Unless 2 = 0 on $\mathbb{C}$, which is clearly false, but it must be explicitly stated to Lean as well.

```
  x : n  , has_eigenpair M  x  vec_im x = 0 := begin
-- We know that    from before.
have H_ : .im = 0,
{ apply M.symm_matrix_real_eigenvalues H_symm  H_eigenvalue },
rcases H_eigenvalue with x, H_nx, H_mul ,
by_cases H_re : vec_re x = 0,
-- 1) I  x  will be used (I corresponds to i from mathematics)
{ use (I  x),
  split,
  -- 1.1) I  x is an eigenvector
  { split,
    -- 1.1.1) I  x  0
    { intro hyp, rw smul_eq_zero at hyp,
      have H_nI : I  0, { exact I_ne_zero },
      tauto },
    -- 1.1.2) M (I  x) =   (I  x)
    { simp only [mul_vec_smul_assoc, H_mul, smul_smul, mul_comm] } },
  -- 1.2) I  x
  { ext i,
    simp only [vec_re, vec_eq_unfold] at H_re,
    simp only [vec_im, algebra.id.smul_eq_mul, I_re, one_mul,
               I_im, zero_mul, mul_im, zero_add, pi.smul_apply],
    exact H_re i } },
-- 2) x + x* will be used
{ use (x + vec_conj x),
  split,
  -- 2.1) x + x* is an eigenvector
  { split,
    -- 2.1.1) x + x*  0
    { intro hyp, exact H_re (vec_conj_add_zero hyp) },
    -- 2.1.2) M (x + x*) =   (x + x*)
    calc M.Coe.mul_vec (x + vec_conj x)
        = M.Coe.mul_vec x + M.Coe.mul_vec (vec_conj x) :
    by { apply mul_vec_add } -- M (x + x*) = M x + M x*
    ... = M.Coe.mul_vec x + vec_conj (M.Coe.mul_vec x) :
    by { rw  M.vec_conj_mul_vec_re x }      -- ... = M x + (M x)*
    ... =   x + vec_conj (  x) :
    by { rw H_mul }                          -- ... =   x + (  x)*
    ... =   x + (conj )  (vec_conj x) :
    by { rw vec_conj_smul }                  -- ... =   x + *  x*
    ... =   x +   (vec_conj x) :
    by { rw conj_of_zero_im H_ }             -- ... =   x +   x*
    ... =   (x + vec_conj x) :
    by { simp only [smul_add] } },           -- ... =   (x + x*)
  -- 2.2) x + x*
  { ext, simp [vec_add_conj_eq_two_re, vec_im, coe_vec_re] } }
end

lemma I_ne_zero : (I : )  0
lemma vec_eq_unfold (x y : n  ) :
   ( i : n, x i) = ( i : n, y i)   i : n, x i = y i
lemma I_re : I.re = 0
```

```
lemma I_im : I.im = 1
lemma one_mul :  a : , 1 * a = a
lemma zero_mul (a : ) : 0 * a = 0
lemma mul_im (z w : ) : (z * w).im = z.re * w.im + z.im * w.re
lemma zero_add (a : ) : 0 + a = a
lemma vec_conj_add_zero {x : n  } (H: x + vec_conj x = 0): vec_re x = 0
lemma mul_vec_add (A : matrix m n ) (x y : n  ) :
  A.mul_vec (x + y) = A.mul_vec x + A.mul_vec y
lemma conj_of_zero_im { : } (H_im : .im = 0) : conj  =
lemma vec_add_conj_eq_two_re (x : n  ) :
   x + vec_conj x = (2 : )  (vec_re x : n  )
lemma coe_vec_re (x : n  ) {i : n} :
   (vec_re x : n  ) i = ((x i).re : )
```

Lemma 4.5 *If* $\boldsymbol{v}$ *and* $\boldsymbol{w}$ *are eigenvectors of a symmetric real-valued matrix* M *with different eigenvalues, then* $\boldsymbol{v}$ *and* $\boldsymbol{w}$ *are orthogonal.*

Proof. Let μ and μ' be the corresponding eigenvalues of the eigenvectors $\mathbf{v}$ and $\mathbf{w}$. It can be shown that (μ - μ') $\mathbf{v}^\intercal\mathbf{w} = 0$ (details in the Lean proof below). Since μ and μ' are distinct, we get that $\mathbf{v}^\intercal\mathbf{w} = 0$ as required.

```
theorem dot_product_neq_eigenvalue_zero (H_symm : symm_matrix M) ( ′: )
(v w : n  ) (H : has_eigenpair M  v) (H : has_eigenpair M ′ w)
(H_ne :  ′) : dot_product v w = 0 := begin
  have key : ( - ′) * dot_product v w = 0,
  calc ( - ′) * dot_product v w
      =  * dot_product v w - ′ * dot_product v w :
  by { apply mul_sub_right_distrib }-- ( - ′)v w = (v w) - ′(v w)
  ... = dot_product (  v) w - dot_product v (′  w) :
  by { simp only [dot_product_smul,
       smul_dot_product] }                    -- ... = (v)w - v(′w)
  ... = dot_product (M.Coe.mul_vec v) w - dot_product v (M.Coe.mul_vec w)
    :
  by { rw [H.2, H.2] }                      -- ... = (M v)w - v(M w)
  ... = dot_product (M.Coe.mul_vec v) w - dot_product (vec_mul v M.Coe) w
    :
  by { rw M.dot_product_mul_vec_vec_mul v w }-- ... = (M v)w - (v M)w
  ... = dot_product (M.Coe.mul_vec v) w - dot_product (vec_mul v M.Coe) w:
  by { rw  symm_matrix_coe M H_symm }       -- ... = (M v)w - (v M)w
  ... = dot_product (M.Coe.mul_vec v) w - dot_product (mul_vec M.Coe v) w
    :
  by { rw  vec_mul_transpose M.Coe v }      -- ... = (M v)w - (M v)w
  ... = 0 :
  by { simp only [sub_self] },              -- ... = 0
  rw mul_eq_zero at key,
  cases key with H_ H_dot,
  { exfalso, rw sub_eq_zero at H_, exact H_ne H_ }, --  - ′ = 0
  { exact H_dot } -- v w = 0
end
```

```
lemma mul_sub_right_distrib (a b c : ) : (a - b) * c = a * c - b * c
lemma dot_product_mul_vec_vec_mul (v w : n  ) :
  dot_product v (mul_vec M.Coe w) = dot_product (vec_mul v M.Coe) w
lemma symm_matrix_coe (H_symm : symm_matrix M) : (M.Coe) = (M.Coe)
lemma vec_mul_transpose (A : matrix m n ) (x : n  ) :
  vec_mul x A = mul_vec A x
```

5. Incidence matrices

Let us fix an undirected simple graph G = (V, E). The **incidence matrix** M of G is the |V| × |E| matrix with M i e = 1 iff edge e is incident to vertex i, and 0 otherwise. To be as general as possible, M is defined to have entries coming from a ring R that is **nontrivial** (it has at least two elements, 1 ≠ 0) and where equality between elements is decidable.

```
universe u
variables {R : Type u} [ring R] [nontrivial R] [decidable_eq R]

namespace simple_graph

universe v
variables {V : Type v} [fintype V] (G : simple_graph V) (R)
    [decidable_rel G.adj] [decidable_eq V]

def inc_matrix : matrix V G.edge_set R
| i e := if (e : sym2 V)  G.incidence_set i then 1 else 0
```

As seen in Section 3.4, the edge set *G.edge_set* for any simple graph G is a set of unordered pairs (*sym2 V*). In our definition we use this particular set as one of the dimensions of the incidence matrix, so Lean must **coerce** this set to the corresponding type. Although we can write expressions like *e : G.edge_set*, Lean internally will attribute the type ↥*G.edge_set* to e, which is a **subtype** of *sym2 V*. The definition of **subtype** in Lean is:

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

Given a set of elements of type α, the correponding subtype has two fields:

- *val*[1] = the corresponding value of the bigger type (here *e.val* is an unordered pair).
- *property* = the proof that the corresponding value is in the set.

5.1 Lemmas for incidence matrices

Lemma 5.1 *For any adjacent vertices i and j, the dot product between M i and M j is 1.*

Proof. We split the proof that Σ e : E, M i e * M j e = 1 into two cases: if e is an edge between i and j, then M i e = M j e = 1 ⇒ M i e * M j e = 1 ; otherwise, e is not incident to at least one of i and j ⇒ at least one of M i e and M j e is 0 ⇒ M i e * M j e = 0. Thus, the sum reduces to the cardinality of the set containing edges that are adjacent to

[1] Equivalent to coercing from ↥*G.edge_set.* to *sym2 V*

both i and j, which contains exactly one element, since i and j are adjacent and we do not allow for multiple edges between the same vertices. Thus, the sum is equal to 1.

```
lemma adj_sum_of_prod_inc_one {i j : V} (H_adj : G.adj i j) :
   (e : G.edge_set), G.inc_matrix R i e * G.inc_matrix R j e = (1 : R) :=
begin
  simp only [inc_matrix_apply, ite_prod_one_zero, sum_boole,
     G.mem_incidence_sets_iff_eq H_adj, val_eq_coe],
  -- ((filter ( (x : (G.edge_set)), x.val = (i, j)) univ).card) = 1
  rw adj_iff_exists_edge_val at H_adj,
  rcases H_adj with e, H_e,
  simp only [H_e, edge_val_equiv],
  -- ((filter ( (x : (G.edge_set)), x = e) univ).card) = 1
  have H : filter ( (x : G.edge_set), x = e) univ = {e},
  { ext, simp only [true_and, mem_filter, mem_univ, mem_singleton] },
  simp only [H, filter_congr_decidable, cast_one, card_singleton]
end
```

- univ is the **finite set** of all elements of a certain type, we can write it explicitly with

  univ : finset α or, as in our case, leave Lean infer the underlying type.

```
lemma inc_matrix_apply {i : V} {e : G.edge_set} :
  G.inc_matrix R i e = if (e : sym2 V)  G.incidence_set i then 1 else 0
lemma inc_matrix_apply {P Q : Prop} [decidable P] [decidable Q] :
  (ite P 1 0) * (ite Q 1 0) = ite (P  Q) (1 : R) 0
lemma sum_boole {s : finset } {p :  Prop} {hp : decidable_pred p} :
  ( x in s, if p x then (1 : ) else (0 : )) = (s.filter p).card
lemma mem_incidence_sets_iff_eq {i j : V} {e : sym2 V} (h : G.adj i j) :
  e  G.incidence_set i  e  G.incidence_set j  e = (i, j)
lemma val_eq_coe {x : subtype p} : x.1 = x -- remember that x.1 = x.val
lemma adj_iff_exists_edge_val {i j : V} :
  G.adj i j   (e : G.edge_set), e.val = (i, j)
lemma edge_val_equiv {e e : G.edge_set} : e.val = e.val  e = e
lemma true_and (a : Prop) : true  a  a
lemma mem_filter {s : finset } {a : } : a  s.filter p  a  s  p a
lemma mem_univ (x : ) : x  (univ : finset )
lemma mem_singleton {a b : } : b  ({a} : finset )  b = a
lemma filter_congr_decidable {} (s : finset ) (p :  Prop)
  (h : decidable_pred p): @filter  p h s = s.filter p
lemma cast_one : ((1 : ) : ) = 1
lemma card_singleton (a : ) : card ({a} : finset ) = 1
```

Lemma 5.2 *For any distinct*[1] *non-adjacent vertices i and j and edge e, M i e*M j e = 0.*

Proof. We either have M i e = 0 or M i e = 1, the former immediately leads to the

[1] This detail, although essential, can easily be missed, but Lean cannot finish the proof without it.

conclusion. The same reasoning applies to M j e, therefore let us consider the non-trivial case when M i e = M j e = 1. These two equalities imply that e is incident to both i and j, therefore e must be an edge between i and j. Since i and j are not the same vertex, we obtain that i and j are adjacent in G, which leads to a contradiction.

```
theorem inc_matrix_prod_non_adj {i j : V} {e : G.edge_set} (Hne : i  j)
(H_non_adj : ň G.adj i j) : G.inc_matrix R i e * G.inc_matrix R j e = 0 :
    =
begin
  by_cases H : G.inc_matrix R i e = 0,
  { rw [H, zero_mul] },
  { rw [inc_matrix_not_zero, inc_matrix_one] at H,
    by_cases H : G.inc_matrix R j e = 0,
    { rw [H, mul_zero] },
    { rw [inc_matrix_not_zero, inc_matrix_one] at H,
      exfalso,
      apply H_non_adj,
      rw [mem_edge_set, G.edge_in_two_incidence_sets Hne H, H],
      exact G.edge_val_in_set } }
end
```

```
lemma inc_matrix_not_zero {i : V} {e : G.edge_set} :
  ň G.inc_matrix R i e = 0  G.inc_matrix R i e = 1
lemma inc_matrix_one {i : V} {e : G.edge_set} :
  G.inc_matrix R i e = 1  e.val  G.incidence_set i
lemma mul_zero (a : ) : a * 0 = 0
lemma mem_edge_set {v w : V} : (v, w)  G.edge_set  G.adj v w
lemma edge_in_two_incidence_sets {i j : V} {e : sym2 V} (H_ne : i  j) :
  e  G.incidence_set i  e  G.incidence_set j  e = (i, j)
lemma edge_val_in_set {e : G.edge_set} : e.val  G.edge_set
```

Lemma 5.3 *Every element of the incidence matrix is idempotent.*

Proof. For any vertex i and edge e, we have two cases: if e is incident to i, $(\text{M i e})^2 = 1^2 = 1 = \text{M i e}$. Otherwise, $(\text{M i e})^2 = 0^2 = 0 = \text{M i e}$.

```
theorem inc_matrix_element_power_id {i : V} {e : G.edge_set} :
  (G.inc_matrix R i e) * (G.inc_matrix R i e) = G.inc_matrix R i e :=
begin
  by_cases H : G.inc_matrix R i e = 1,
  { rw [H, mul_one] },
  { rw inc_matrix_not_one at H, rw [H, mul_zero] }
end
```

```
lemma mul_one :  a : , a * 1 = a
lemma inc_matrix_not_one {i : V} {e : G.edge_set} :
  ň G.inc_matrix R i e = 1  G.inc_matrix R i e = 0
```

Lemma 5.4 *The degree of any vertex i is equal to the sum of elements from its corresponding row in the incidence matrix.*

Proof. By definition, the degree of a vertex i is equal to the number of edges that are incident to it. Furthermore, each edge e that is incident to i will have M i e = 1, so the sum over the entire row i will return the degree of i.

```
theorem degree_equals_sum_of_incidence_row {i : V} :
  (G.degree i : R) =  (e : G.edge_set), G.inc_matrix R i e :=
begin
  rw [inc_matrix_def, card_incidence_set_eq_degree],
  simp only [sum_boole, nat.cast_inj,
             fintype.card_coe_filter (G.incidence_set_subset i)],
end
```

```
lemma inc_matrix_def : G.inc_matrix R = -- ite = if..then..else
   i e, ite ((e : sym2 V)  G.incidence_set i) 1 0
lemma card_incidence_set_eq_degree :
  fintype.card (G.incidence_set v) = G.degree v
lemma cast_inj {m n : } : (m : R) = n  m = n
lemma fintype.card_coe_filter { : Sort*} {s t : set } (h : s  t) :
  fintype.card s = finset.card (finset.filter ( (x : t), (x : )  s)
    finset.univ)
```

5.2 Towards oriented graphs : Orientations

In the next section we will prove that the Laplacian matrix can be decomposed into a product of matrices, and for this we need to define the notion of **orientation** in graph theory and then to derive the **oriented graph** of an undirected simple graph.

An **orientation**[1] is an assignment of a direction to the edges of an undirected graph. Each edge will become unidirectional and we will call the starting vertex the **head** of the edge and the ending vertex the **tail** of the edge. We will define this in Lean as:

```
@[ext]
structure orientation (G : simple_graph V) :=
(head : G.edge_set  V)
(tail : G.edge_set  V)
(consistent (e : G.edge_set) : e.val = (head(e), tail(e)))
```

The *consistent* property asserts that for each edge, we assign the *head* and the *tail* such that together they form the edge (making sure they do not point to the same vertex).

Given an undirected graph G and an orientation o on its edges, we can then define the **oriented graph** with respect to G and o as the directed graph that is formed by orienting all edges in G according to o. Notice that every oriented graph is a directed graph, but the reverse direction is not true, since oriented graphs do not allow cycles of size 2.

[1] https://en.wikipedia.org/wiki/Orientation_(graph_theory)

5.3 Oriented incidence matrix

The **oriented incidence matrix** N_o can now be defined in terms of an undirected graph G and an orientation o as the |V| × |E| matrix with:

- N_o i e = 1, if i is the head vertex of edge e in G
- N_o i e = -1, if i is the tail vertex of edge e in G
- N_o i e = 0, otherwise

```
def oriented_inc_matrix (o : orientation G) : matrix V G.edge_set R :=
 i e, if i = o.head e then (1 : R) else (if i = o.tail e then -1 else 0)
```

Lemma 5.5 *For each vertex i, edge e and orientation o,* $(N_o\ i\ e)^2 = M\ i\ e$.

Proof. We distinguish three cases:

1. i is the head of e ⇒ N_o i e = M i e = 1 ⇒ the relation becomes $1^2 = 1$
2. i is the tail of e ⇒ N_o i e = -1 and M i e = 1 ⇒ the relation becomes $(-1)^2 = 1$
3. e is not incident to i ⇒ N_o i e = M i e = 0 ⇒ the relation becomes $0^2 = 0$

And all the above relations clearly hold under any nontrivial ring R.

```
theorem oriented_inc_matrix_elem_squared {i : V} {e : G.edge_set} :
  G.oriented_inc_matrix R o i e * G.oriented_inc_matrix R o i e =
    G.inc_matrix R i e :=
begin
  by_cases H_head : i = o.head e,
  -- Case 1
  { rw [G.oriented_inc_matrix_head R H_head, H_head,
        mul_one, eq_comm, inc_matrix_one],
    exact G.incidence_set_orientation_head },
  { by_cases H_tail : i = o.tail e,
    -- Case 2
    { rw [G.oriented_inc_matrix_tail R H_tail, H_tail, mul_neg_eq_neg_-
   mul_symm, mul_one, neg_neg, eq_comm, inc_matrix_one],
      exact G.incidence_set_orientation_tail },
    -- Case 3
    { rw [(G.oriented_inc_matrix_zero R).mpr H_head, H_tail,
          mul_zero, eq_comm, inc_matrix_zero],
      exact G.not_inc_set_orientation H_head H_tail } }
end
```

- .mp/.mpr on lemmas of the form $a \leftrightarrow b$ will retrieve $a \to b$ and $b \to a$, respectively.

```
lemma oriented_inc_matrix_head {i : V} {e : G.edge_set}
  (H_head : i = o.head e) : G.oriented_inc_matrix R o i e = 1
lemma incidence_set_orientation_head {e : G.edge_set} :
  e.val  G.incidence_set (o.head e)
lemma oriented_inc_matrix_tail {i : V} {e : G.edge_set}
  (H_tail : i = o.tail e) : G.oriented_inc_matrix R o i e = -1
lemma mul_neg_eq_neg_mul_symm (a b : ) : a * - b = - (a * b)
lemma neg_neg (a : ) : - -a = a
lemma incidence_set_orientation_tail {e : G.edge_set} :
  e.val  G.incidence_set (o.tail e)
lemma oriented_inc_matrix_zero {i : V} {e : G.edge_set} :
  G.oriented_inc_matrix R o i e = 0  i  o.head e  i  o.tail e
lemma inc_matrix_zero {i : V} {e : G.edge_set} :
  G.inc_matrix R i e = 0  e.val  G.incidence_set i
lemma not_inc_set_orientation {i : V} {e : G.edge_set}
  (H_head : i  o.head e) (H_tail : i  o.tail e) :
  e.val  G.incidence_set i
```

Lemma 5.6 *For any two adjacent vertices i and j and edge e, there are two cases: if e is between i and j in G, then N_o i e * N_o j e = -1, otherwise the product will be 0.*

Proof. We split the proof in two (exhaustive) cases:

1. e is the edge between i and j in G (it exists since i and j are adjacent)
   (a) i is the head of e ⇒ j is the tail of e ⇒ N_o i e * N_o j e = 1 * (-1) = -1.
   (b) i is the tail of e ⇒ j is the head of e ⇒ N_o i e * N_o j e = (-1) * 1 = -1.
2. e is not the edge between i and j in G
   (a) i and j are not the head of e ⇒ at least one of them is not the tail of e[1] ⇒ at least one of N_o i e and N_o j e is 0 ⇒ their product is 0
   (b) e is not incident to i ⇒ N_o i e = 0 ⇒ the product is 0
   (c) e is not incident to j ⇒ N_o j e = 0 ⇒ the product is 0
   (d) i and j are not the tail of e ⇒ at least one of them is not the head of e ⇒ at least one of N_o i e and N_o j e is 0 ⇒ their product is 0

```
theorem oriented_inc_matrix_prod_of_adj {i j : V} {e : G.edge_set}
(H_adj : G.adj i j) : G.oriented_inc_matrix R o i e *
   G.oriented_inc_matrix R o j e = ite (e.val = (i, j)) (-1) 0 :=
begin
  by_cases H_e : e.val = (i, j),
  -- 1) e is the edge between i and j
```

[1] If both would be, they would be equal, contradicting the fact that they are adjacent.

```
  { rw [H_e, if_pos rfl],
    rw [o.consistent e, sym2.eq_iff] at H_e,
    rcases H_e with (H_head_i, H_tail_j | H_head_j, H_tail_i),
    { rw [G.oriented_inc_matrix_head R H_head_i.symm,
          G.oriented_inc_matrix_tail R H_tail_j.symm,
          mul_neg_eq_neg_mul_symm, mul_one] },
    { rw [G.oriented_inc_matrix_head R H_head_j.symm,
          G.oriented_inc_matrix_tail R H_tail_i.symm, mul_one] } },
  -- 2) e is not the edge between i and j
  { simp only [H_e, if_false],
    rw [o.consistent e, eq_iff, decidable.not_or_iff_and_not ] at H_e,
    repeat { rw decidable.not_and_iff_or_not at H_e },
    rcases H_e with (H_head_i | H_tail_j), (H_head_j | H_tail_i),
    -- 2.a) both i and j are not the head of e
    { have H_tail : o.tail e  i  o.tail e  j,
      { by_contradiction h,
        rw [decidable.not_or_iff_and_not, not_not, not_not] at h,
        rcases h with h_i, h_j, rw h_i at h_j,
        exact G.ne_of_adj H_adj h_j },
      cases H_tail with H_tail_i H_tail_j,
      -- 2.a.1) i is not the tail of e
      { rw [(G.oriented_inc_matrix_zero R).mpr
            ne.symm H_head_i, ne.symm H_tail_i, zero_mul] },
      -- 2.a.2) j is not the tail of e
      { rw [(G.oriented_inc_matrix_zero R).mpr
            ne.symm H_head_j, ne.symm H_tail_j, mul_zero] } },
    -- 2.b) i is neither the head of e nor its tail
    { rw [(G.oriented_inc_matrix_zero R).mpr
          ne.symm H_head_i, ne.symm H_tail_i, zero_mul] },
    -- 2.c) j is neither the head of e nor its tail
    { rw [(G.oriented_inc_matrix_zero R).mpr
          ne.symm H_head_j, ne.symm H_tail_j, mul_zero] },
    -- 2.d) both i and j are not the tail of e
    { have H_head : o.head e  i  o.head e  j,
      { by_contradiction h,
        rw [decidable.not_or_iff_and_not, not_not, not_not] at h,
        rcases h with h_i, h_j, rw h_i at h_j,
        exact G.ne_of_adj H_adj h_j },
      cases H_head with H_head_i H_head_j,
      -- 2.d.1) i is not the head of e
      { rw [(G.oriented_inc_matrix_zero R).mpr
            ne.symm H_head_i, ne.symm H_tail_i, zero_mul] },
      -- 2.d.2) j is not the head of e
      { rw [(G.oriented_inc_matrix_zero R).mpr
            ne.symm H_head_j, ne.symm H_tail_j, mul_zero] } } }
end
```

```
lemma eq_iff {x y z w : } :
  (x, y) = (z, w)  (x = z  y = w)  (x = w  y = z)
lemma if_false { : Sort u} (t e : ) : (ite false t e) = e
lemma not_or_iff_and_not (p q : Prop) : ň (p  q)  ň p  ň q
lemma not_and_iff_or_not (p q : Prop) : ň (p  q)  ň p  ň q
lemma not_not : ňňa  a
```

```
lemma ne_of_adj {a b : V} (hab : G.adj a b) : a  b
```

Lemma 5.7 *For any distinct non-adjacent vertices i, j and edge e, N_o i e * N_o j e = 0.*

Proof. We distinguish the following cases:

1. N_o i e = 0 or N_o j e = 0 ⇒ the product is 0
2. N_o i e and N_o j e are both non-zero ⇒ e is incident to both i and j:
   (a) i and j are the head of e ⇒ contradiction to the fact that they are distinct
   (b) i is the head, j is the tail ⇒ contradiction to the fact that they are non-adjacent
   (c) i is the tail, j is the head ⇒ contradiction to the fact that they are non-adjacent
   (d) i and j are the tail of e ⇒ contradiction to the fact that they are distinct

Therefore, the product will always be 0.

```
theorem oriented_inc_matrix_prod_non_adj {i j : V} {e : G.edge_set}
(H_ij : i  j) (H_not_adj : ň G.adj i j) :
  G.oriented_inc_matrix R o i e * G.oriented_inc_matrix R o j e = 0 :=
begin
  by_cases H : G.oriented_inc_matrix R o i e = 0,
  -- Case 1.a
  { rw [H, zero_mul] },
  { by_cases H : G.oriented_inc_matrix R o j e = 0,
    -- Case 1.b
    { rw [H, mul_zero] },
    { rcases ((G.oriented_inc_matrix_non_zero R).mp H)
             with (H_head_i | H_tail_i) ;
      rcases ((G.oriented_inc_matrix_non_zero R).mp H)
             with (H_head_j | H_tail_j),
      -- Case 2.a
      { rw [H_head_i, H_head_j] at H_ij, tauto },
      -- Case 2.b
      { exfalso, apply H_not_adj,
        rw [H_head_i, H_tail_j, mem_edge_set, o.consistent e],
        simp only [subtype.coe_prop, subtype.val_eq_coe] },
      -- Case 2.c
      { exfalso, apply H_not_adj, apply (G.edge_symm i j).mpr,
        rw [H_tail_i, H_head_j, mem_edge_set, o.consistent e],
        simp only [subtype.coe_prop, subtype.val_eq_coe] },
      -- Case 2.d
      { rw [H_tail_i, H_tail_j] at H_ij, tauto } } }
end
```

```
lemma oriented_inc_matrix_non_zero {i : V} {e : G.edge_set} :
  ň G.oriented_inc_matrix R o i e = 0  i = o.head e  i = o.tail e
lemma coe_prop {S : set } (a : {a // a  S}) : a  S := a.property
```

```
lemma val_eq_coe {x : subtype p} : x.1 = x -- x.1 is equivalent to x.val
lemma edge_symm (u v : V) : G.adj u v  G.adj v u
```

Lemma 5.8 *Let $\boldsymbol{x}$ be a vector indexed on V with values from R. Then, $(\boldsymbol{x}^\intercal\ N_o)_e = \boldsymbol{x}$ $head_o(e)$ - $\boldsymbol{x}$ $tail_o(e)$, where the functions $head_o$ and $tail_o$ take an edge e and return the corresponding vertex according to the orientation o.*

Proof. The LHS is equivalent to Σ (i : V), **x** i * N_o i e. On every column e of the oriented incidence matrix, there are exactly two entries that are non-zero, which correspond to the two ends of the edge: the entry with row $head_o$(e) will have a 1 and the entry with row $tail_o$(e) will have a (-1). Thus, the LHS becomes **x** $head_o$(e) - **x** $tail_o$(e), as desired.

```
theorem vec_mul_oriented_inc_matrix {o : orientation G} (x : V  R)
(e : G.edge_set) :
  vec_mul x (G.oriented_inc_matrix R o) e = x (o.head e) - x (o.tail e) :
    =
begin
  simp only [vec_mul, dot_product, oriented_inc_matrix, mul_ite,
             mul_one, mul_neg_eq_neg_mul_symm, mul_zero],
  rw [sum_ite, sum_ite, sum_filter, sum_ite_eq', sum_const_zero,
             add_zero, filter_filter],
  simp only [mem_univ, if_true],
  have key :
    filter ( (a : V), ňa = o.head e  a = o.tail e) univ = {o.tail e},
  { ext,
    simp only [mem_filter, mem_singleton, true_and,
               and_iff_right_iff_imp, mem_univ],
    rintro rfl,
    exact ne.symm (G.head_neq_tail) },
  rw [key, sum_singleton],
  ring_nf
end
```

```
lemma mul_ite {} (P : Prop) (a b c : ) :
  a * (if P then b else c) = if P then a * b else a * c
lemma sum_ite {s : finset } {p :   Prop} (f g :   ) :
  ( x in s, if p x then f x else g x) =
  ( x in s.filter p, f x) + ( x in s.filter ( x, ň p x), g x)
lemma sum_filter (p :   Prop) (f :   ) :
  ( a in s.filter p, f a) = ( a in s, if p a then f a else 0)
lemma sum_ite_eq' (s : finset ) (a : ) (b :   ) :
  ( x in s, (ite (x = a) (b x) 1)) = ite (a  s) (b a) 0
lemma sum_const_zero {} {s : finset } : ( x in s, (0 : )) = 0
lemma add_zero :  a : , a + 0 = a
lemma filter_filter (s : finset ) :
  (s.filter p).filter q = s.filter (a, p a  q a)
lemma if_true { : Sort u} (t e : ) : (ite true t e) = t
lemma and_iff_right_iff_imp {a b : Prop} : ((a  b)  b)  (b  a)
lemma head_neq_tail {e : G.edge_set} : o.head(e)  o.tail(e)
```

```
lemma sum_singleton : ( x in (singleton a), f x) = f a
```

- ring_nf simplifies expressions in the language of commutative (semi)rings, which rewrites all ring expressions into a normal form and then checks if they are equal.

6. Laplacian matrices

Let us fix an undirected simple graph G = (V, E). The **Laplacian matrix**[1] L is the |V| × |V| matrix with

- L i j = degree(i), if i = j
- L i j = - A i j, otherwise (A is the adjacency matrix of G)

To be as general as possible, we will define M to have entries in a commutative ring R (combining the ring property of R from incidence matrices with the commutative property of R from the adjacency matrices) that is nontrivial and with decidable equality relation.

```
universes u v
variables {V : Type u} [fintype V] [decidable_eq V]
variables {R : Type v} [comm_ring R] [nontrivial R] [decidable_eq R]

namespace simple_graph

variables (G : simple_graph V) (R) [decidable_rel G.adj]

def laplace_matrix : matrix V V R
| i j := if i = j then G.degree i else - G.adj_matrix R i j
```

Most of the lemmas presented in this section can also be found at [8].

Lemma 6.1 *For every vertex $i \in V$, degree(i) = the sum of elements from row i of A.*

Proof. Using the definition of A, the sum of its elements from a particular row i is composed of ones (for entries (i, j) where i and j are adjacent) and of zeroes (for entries (i, j) where i and j are not adjacent). Therefore, the sum on row i is equal to the total number of vertices that are adjacent to i, which is precisely the degree of i.

```
theorem degree_eq_sum_of_adj_matrix_row { : Type*} [semiring ] {i : V} :
  (G.degree i : ) =  (j : V), G.adj_matrix  i j :=
by { rw [mul_one (G.degree i : )],
     simp only [adj_matrix_mul_vec_const_apply, mul_vec,
                dot_product, boole_mul, adj_matrix_apply] }
```

```
lemma adj_matrix_mul_vec_const_apply {r : R} {v : } :
  (G.adj_matrix R).mul_vec (function.const _ r) v = G.degree v * r
lemma boole_mul {} [semiring ] (P : Prop) (a : ) :
  (if P then 1 else 0) * a = if P then a else 0
lemma adj_matrix_apply (v w : ) :
  G.adj_matrix R v w = if (G.adj v w) then 1 else 0
```

[1] https://en.wikipedia.org/wiki/Laplacian_matrix

6.1 Lemmas for Laplacian matrices

Lemma 6.2 *The Laplacian matrix L is symmetric.*

Proof. Our goal is to show that L i j = L j i for any two distinct vertices i and j. Using the definition of the Laplacian matrix, we restrict the proof to A i j = A j i. This clearly holds from the definition of the adjacency matrix A (adjacency is bidirectional).

```
theorem transpose_laplace_matrix :
   (G.laplace_matrix R) = G.laplace_matrix R :=
begin
  ext i j,
  by_cases H : (i = j),
  { simp only [H, transpose_apply] },
  { rw [transpose_apply, G.laplace_matrix_neq R H,
        G.laplace_matrix_neq R (ne.symm H)],
    simp [edge_symm] }
end
```

```
lemma transpose_apply (M : matrix m n ) (i j) :
   M.transpose j i = M i j
lemma laplace_matrix_neq {i j : V} (H_neq : i  j) :
  G.laplace_matrix R i j = - G.adj_matrix R i j
```

Lemma 6.3 *For every vertex $i \in V$, the sum of elements from row i of L is 0.*

Proof. The sum is equal to L i i + Σ (j : V \ {i}), A i j = degree(i) - degree(i)[1] = 0.

```
theorem sum_of_laplace_row_equals_zero {i : V} :
   (j : V), G.laplace_matrix R i j = 0 :=
begin
  rw [sum_eq_add_sum_diff_singleton (mem_univ i), laplace_matrix_eq],
  simp only [laplace_matrix_apply, sum_ite, filter_eq_neq_empty,
             filter_id, adj_matrix_apply],
  rw [sum_neg_distrib, sum_boole, sum_const, card_empty,
      zero_smul, zero_add, degree_eq_sum_of_adj_matrix_row],
  have H: filter ( (x:V), G.adj i x) (univ\{i}) = filter (G.adj i) univ,
  { ext,
    simp only [true_and, mem_filter, mem_sdiff,
               and_iff_right_iff_imp, mem_univ, mem_singleton],
    intro hyp,
    exact ne.symm (G.ne_of_adj hyp) },
  simp only [H, adj_matrix_apply, sum_boole,
             add_right_neg, eq_self_iff_true],
end
```

```
lemma sum_eq_add_sum_diff_singleton {s : finset } {i : } (h : i  s)
  (f :   ) :  x in s, f x = f i +  x in s \ {i}, f x
lemma laplace_matrix_eq {i j : V} (H_eq : i = j) :
  G.laplace_matrix R i j = G.degree i
```

[1] Using Lemma 6.1.

```
lemma laplace_matrix_apply {i j : V} :
  G.laplace_matrix R i j = ite (i = j) (G.degree i) (- G.adj_matrix R i
    j)
lemma filter_eq_neq_empty {i : V} : filter (eq i) (univ \ {i}) =
lemma filter_id {i : V} :
  filter ( (x : V), ňi = x) (univ \ {i}) = (univ \ {i})
lemma sum_neg_distrib : ( x in s, - (f x)) = - ( x in s, f x)
lemma sum_const (b : ) : ( x in s, b) = s.card  b
lemma card_empty : card ( : finset ) = 0
lemma zero_smul (m : M) : (0 : R)  m = 0
lemma mem_sdiff {a : } {s s : finset } :
  a  s \ s  a  s  a  s
lemma add_right_neg (a : ) : a + -a = 0
lemma eq_self_iff_true { : Sort u} (a : ) : (a = a)  true
```

Lemma 6.4 (Laplacian decomposition) *For any orientation o, $L = N_o\ N_o^\intercal$.*

Proof. First, let us prove the equality on the diagonal: for any vertex i, L i i = degree(i). Now, (N_o $N_o^\intercal$) i i = Σ (e : E), (N_o i e)^2. Using Lemma 5.5, the sum becomes Σ (j : V), M i e and then Lemma 5.4 closes the goal.

Secondly, to prove the equality for non-diagonal elements, let us take two distinct vertices i and j. Our goal now is to prove that L i j = (N_o $N_o^\intercal$) i j. From the definition of the Laplacian matrix, the LHS becomes - A i j. The RHS can be rewritten as Σ (e : E), N_o i e * N_o j e. We can consider now two cases:

1. i and j are adjacent ⇒ the LHS becomes -1 and from Lemma 5.6 the RHS becomes Σ (e : E), if e is the edge between i and j then (-1) else 0. Since there is exactly one edge between i and j, the RHS sum will be -1 as required.

2. i and j are not adjacent ⇒ the LHS becomes 0 and from Lemma 5.7 the RHS becomes a sum of zeroes, which reduces to 0, as required.

At case 1, we will require a new definition of a function which inputs two vertices i and j and a proof that they are adjacent in G and returns the bidirectional edge between them.

```
def edge_from_verts (i j : V) (H_adj : G.adj i j) : G.edge_set :=
  (i, j), G.mem_edge_set.mpr H_adj
```

```
theorem laplace_decomposition (o : orientation G) : G.laplace_matrix R =
  G.oriented_inc_matrix R o  (G.oriented_inc_matrix R o) :=
begin
  ext i j,
  by_cases H_ij : i = j,
  -- diagonal : i = j
```

```
  { rw [G.laplace_matrix_eq R H_ij, mul_apply, H_ij,
    G.degree_equals_sum_of_incidence_row R],
    simp only [transpose_apply, G.oriented_inc_matrix_elem_squared R] },
  -- non-diagonal : i  j
  { rw [G.laplace_matrix_neq R H_ij, mul_apply],
    by_cases H_adj : G.adj i j,
    -- Case 1 : i and j are adjacent
    { simp only [G.adj_matrix_adj R H_adj, transpose_apply,
                 G.oriented_inc_matrix_prod_of_adj R H_adj],
      have key :  (e : G.edge_set),
        ite (e.val = (i, j)) (-1 : R) 0 = - ite (e.val = (i, j)) 1 0,
      { intro e,
        convert (apply_ite ( x : R, -x) (e.val = (i, j)) 1 0).symm,
        rw neg_zero },
      have sum :  (e : G.edge_set), ite (e.val = (i, j)) (-1 : R) 0 =
                  (e : G.edge_set), - ite (e.val = (i, j)) (1 : R) 0,
      { simp only [key] },
      rw [sum, sum_hom, neg_inj, sum_boole],
      have key : filter ( (e : G.edge_set), e.val = (i, j)) univ =
                  {G.edge_from_verts i j H_adj},
      { ext,
        simp only [true_and, mem_filter, mem_univ, mem_singleton],
        rw G.edge_from_verts_iff H_adj },
      rw key,
      simp only [nat.cast_one, card_singleton] },
    -- Case 2 : i and j are not adjacent
    { simp only [G.adj_matrix_not_adj R H_adj, transpose_apply,
                 G.oriented_inc_matrix_prod_non_adj R H_ij H_adj,
                 sum_const_zero, neg_zero] } }
end

lemma mul_apply {M : matrix l m } {N : matrix m n } {i k} :
  (M  N) i k =  j, M i j * N j k
lemma adj_matrix_adj {i j : V} (H_adj : G.adj i j) :
  G.adj_matrix R i j = 1
lemma apply_ite {  : Sort*} (f :   ) (P : Prop) (x y : ) :
  f (ite P x y) = ite P (f x) (f y)
lemma neg_zero : -0 = (0 : )
lemma sum_hom (s : finset ) {f :   } (g :   ):
  ( x in s, g (f x)) = g ( x in s, f x)
lemma neg_inj : -a = -b  a = b
lemma edge_from_verts_iff {i j : V} {e : G.edge_set} (H_adj : G.adj i j)
   :
  e = G.edge_from_verts i j H_adj  e.val = (i, j)
lemma adj_matrix_not_adj {i j : V} (H_not_adj :
  ň G.adj i j) : G.adj_matrix R i j = 0
```

The convert e tactic takes the current goal and creates all the subgoals required in order to transform the goal in the expression denoted by e. It replies with all the tasks remaining to perform in order to solve the current goal using e.

Lemma 6.5 (Laplacian quadratic form) *L is a quadratic form: for any real-valued vector* $\boldsymbol{x}$ *and orientation o,* $\boldsymbol{x}\ L\ \boldsymbol{x}^\intercal = \Sigma\ (e : E),\ (\boldsymbol{x}\ head_o(e) - \boldsymbol{x}\ tail_o(e))^2$.

Proof. Using Lemma 6.4, we can rewrite $\mathbf{x}\ \mathrm{L}\ \mathbf{x}^\intercal$ as $\mathbf{x}\ (\mathrm{N_o}\ \mathrm{N_o}^\intercal)\ \mathbf{x}^\intercal$, which in turn is $(\mathbf{x}^\intercal\ \mathrm{N_o})\ (\mathrm{N_o}^\intercal\ \mathbf{x})$ (details in the Lean proof below). Using Lemma 5.8, we reach the desired result $\Sigma\ (\mathrm{e} : \mathrm{E}),\ (\mathbf{x}\ \mathrm{head_o(e)} - \mathbf{x}\ \mathrm{tail_o(e)})^2$.

```
theorem laplace_quadratic_form {o : orientation G} (x : V  R) :
  dot_product (vec_mul x (G.laplace_matrix R)) x =
   e : G.edge_set, (x (o.head e) - x (o.tail e)) ^ 2 :=
by calc dot_product (vec_mul x (G.laplace_matrix R)) x
      = dot_product (vec_mul x (G.oriented_inc_matrix R o
                    (G.oriented_inc_matrix R o))) x :
                       -- x  L  x = x  (N(o)  N(o))  x
   by { rw laplace_decomposition }
   ... = dot_product (vec_mul (vec_mul x (G.oriented_inc_matrix R o))
                    (G.oriented_inc_matrix R o)) x :
   by { rw vec_mul_vec_mul }  -- ... = (x  N(o))  N(o)  x
   ... = dot_product( j,dot_product(vec_mul x(G.oriented_inc_matrix R o))
                    ( i, (G.oriented_inc_matrix R o) i j)) x :
   by { congr' }
   ... = dot_product (vec_mul x (G.oriented_inc_matrix R o))
     ( (e : G.edge_set), dot_product ((G.oriented_inc_matrix R o) e) x) :
   by { rw dot_product_assoc }
   ... = dot_product (vec_mul x (G.oriented_inc_matrix R o))
                    ((G.oriented_inc_matrix R o).mul_vec x) :
   by { congr' }               -- ... = (x  N(o))  (N(o)  x)
   ... = dot_product (vec_mul x (G.oriented_inc_matrix R o))
                    (vec_mul x (G.oriented_inc_matrix R o)) :
   by { rw mul_vec_transpose } -- ... = (x  N(o))  (x  N(o))
   ...  =  e : G.edge_set, (x (o.head e) - x (o.tail e)) ^ 2 :
   by { simp only [dot_product, vec_mul_oriented_inc_matrix],
        ring_nf }              -- ... =  e,(x head(e)-x tail(e))^2
```

```
lemma vec_mul_vec_mul (v : m  ) (M : matrix m n ) (N : matrix n o ) :
  vec_mul (vec_mul v M) N = vec_mul v (M  N)
```

The congr' n tactic breaks down the goal (which is generally an equality between terms) into equality of sub-terms depending on the level of recursive applications n (in case n is omitted, its implicit value is 1) . For instance, when proving *f (g (x + y)) = f (g (y + x))*, *congr'* produces the goals $x = y$ and $y = x$, whereas *congr' 2* produces the goal $x + y = y + x$, so n affects the depth of dividing the terms into sub-terms.

6.2 Signless Laplacian matrix

Let G = (V, E) be an undirected simple graph. The **Signless Laplacian matrix** Q is defined as the |V| × |V| matrix with

- Q i j = degree(i), if i = j
- Q i j = A i j, otheriwse

It mimics the behaviour of the Laplacian matrix, but it contains only positive elements.

```
def signless_laplace_matrix : matrix V V R
| i j := if i = j then G.degree i else G.adj_matrix R i j
```

Lemma 6.6 *Q = A + D, where D is the diagonal matrix containing the degrees of vertices in G.*

Proof. Let i and j be two vertices:

1. if i = j, then Q i j = D i j = degree(i), whereas A i j = 0 ⇒ Q i j = A i j + D i j
2. if i ≠ j, then Q i j = A i j, whereas D i j = 0 ⇒ Q i j = A i j + D i j

```
lemma signless_laplace_eq_degree_plus_adj : G.signless_laplace_matrix R =
  G.adj_matrix R + diagonal ( v, G.degree v) :=
begin
  ext,
  by_cases H : (i = j),
  -- Case 1
  { rw [G.signless_laplace_matrix_eq R H, dmatrix.add_apply,
        G.adj_matrix_eq R H, zero_add, H, diagonal_apply_eq] },
  -- Case 2
  { rw [G.signless_laplace_matrix_neq R H, dmatrix.add_apply,
        diagonal_apply_ne H, add_zero] }
end
```

The function *diagonal* takes a vector **d** and returns the diagonal matrix that has on the diagonal the entries of **d** in order.

```
def diagonal (d : n  ) : matrix n n  :=
    i j, if i = j then d i else 0
lemma signless_laplace_matrix_eq {i j : V} (H_eq : i = j) :
  G.signless_laplace_matrix R i j = G.degree i
lemma dmatrix.add_apply (M N : dmatrix m n ) (i j) :
  (M + N) i j = M i j + N i j
lemma adj_matrix_eq {i j : V} (H_eq : i = j) : G.adj_matrix R i j = 0
lemma diagonal_apply_eq {d : n  } (i : n) : (diagonal d) i i = d i
lemma signless_laplace_matrix_neq {i j : V} (H_neq : i  j) :
  G.signless_laplace_matrix R i j = G.adj_matrix R i j
lemma diagonal_apply_ne {d : n  } {i j : n} (h : i  j) :
  (diagonal d) i j = 0
```

Lemma 6.7 (Signless Laplacian decomposition) $Q = M\ M^\intercal$.

Proof. Let us prove the equality on the diagonal first: for a vertex i, Q i i = degree(i), whereas (M $M^\intercal$) i i = Σ (e : E), (M i e)^2. Using Lemma 5.3, the RHS can be reduced to Σ (e : E), M i e, which according to Lemma 5.4 is equal to degree(i).

Secondly, to prove the equality for non-diagonal elements, let us take two distinct vertices i and j. Our goal now is to prove that Q i j = (M $M^\intercal$) i j. By the definition of Q, the LHS will be A i j, whereas the RHS can be rewritten as Σ (e : E), (M i e) * (M j e). We can consider now two cases:

1. i and j are adjacent ⇒ the LHS becomes 1 and from Lemma 5.1 the RHS becomes 1 as well, as required

2. i and j are not adjacent ⇒ the RHS becomes 0 and from Lemma 5.2 the RHS becomes a sum of zeroes, which is equivalent to 0, as required.

```
theorem signless_laplace_decomposition :
  G.signless_laplace_matrix R = G.inc_matrix R  (G.inc_matrix R) :=
begin
  ext,
  by_cases H_ij : i = j,
  { rw [signless_laplace_eq_degree_plus_adj, dmatrix.add_apply,
        G.adj_matrix_eq R H_ij, zero_add, mul_apply],
    simp only [H_ij, diagonal_apply_eq,
               degree_equals_sum_of_incidence_row,
               transpose_apply, inc_matrix_element_power_id] },
  { rw [signless_laplace_eq_degree_plus_adj, dmatrix.add_apply,
        diagonal_apply_ne H_ij, add_zero, mul_apply],
    by_cases H_adj : G.adj i j,
    { simp only [G.adj_matrix_adj R H_adj, transpose_apply,
                 G.adj_sum_of_prod_inc_one R H_adj] },
    { simp only [G.adj_matrix_not_adj R H_adj, transpose_apply,
                 G.inc_matrix_prod_non_adj R H_ij H_adj, sum_const_zero]
    } }
end
```

7. Conclusions

7.1 Summary

The project has largely achieved the desired goals outlined in the project description. We have gradually introduced the reader to the Lean background and concepts needed to build a reliable and backwards compatible documentation in domains like linear algebra and graph theory. We managed to go through important, but not yet developed areas such as the properties of the eigenvalues and eigenvectors of symmetric real-valued matrices, contributing (for instance) with helper lemmas to the matrix-vector multiplication API. We can now add two key concepts from graph theory in the library of *mathlib*: the incidence and Laplacian matrices, together with key theorems about their behaviour which will certainly be useful for future development.

7.2 Reflections

One important lesson learnt from this project is the necessity of formal verification of as many mathematical concepts and statements as possible. There were various occasions when the author discovered that the majority of sources used with the intention to prove several lemmas had missed crucial details (corner cases) or had added unnecessary restrictions. The author also became aware throughout this project of the importance of rigorously motivating the use of mathematically-overloaded operations (e.g multiplication, where we distinguished between scalar-vector, vector-matrix, matrix-vector instances and also between real and complex cases) and providing APIs even for the most common-sense uses of them, in order to facilitate expanding to more complicate statements and also to avoid ambiguity.

7.3 Limitations

Naturally, a mathematical proof can follow different styles and ideas in order to achieve a certain goal. They can differ considerably in length, clarity, re-usability and they can use a wide variety of already established concepts. Mostly, Lean proofs are based on equivalent mathematical demonstrations, but they must adopt the ones that maximally use the already existing tools from the main library and this can bring challenges on its own. One goal of this project was to avoid creating unnecessary helper lemmas which would subsequently be used just for the current goal, but this greatly depends on the solution approach and on the current available APIs in Lean. On the other hand, creating new helping lemmas can uncover gaps in the core libraries, which consequently was the source of some small contributions of the project (e.g matrix-vector multiplication operations).

7.4 Future directions

The intention of the project was to establish the basis for the described key mathematical concepts. One extension in Section 4 can be to describe and implement the eigendecomposition of a symmetric matrix, together with concepts such as positive (semi)definiteness and the Perron-Frobenius theorem. Another extension can appear in Section 6, where the spectrum of an undirected simple graph can be defined, based on the work from Section 4, together with implications on bipartite graphs or on the chromatic numbers. Extensions exist as long as there exist mathematical concepts emerging from the existing libraries.

References

[1] Lean official website. https://leanprover-community.github.io/

[2] Lean core library *mathlib*. https://leanprover-community.github.io/mathlib_docs/

[3] Lovász, László. *Eigenvalues of graphs*. Citeseer 2007, p. 1-7.

[4] Brouwer, Andries E and Haemers, Willem H. *Spectra of graphs*. Springer Science & Business Media 2011, p. 1-3.

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

[6] Jeremy Avigad, Kevin Buzzard, Robert Y. Lewis, Patrick Massot. *Mathematics in Lean*. https://leanprover-community.github.io/mathematics_in_lean/mathematics_in_lean.pdf

[7] Jonathan Kelner. *Linear algebra review, adjacency and Laplacian matrices associated with a graph, example Laplacians*. https://ocw.mit.edu/courses/mathematics/18-409-topics-in-theoretical-computer-science-an-algorithmists-toolkit-fall-2009/lecture-notes/MIT18_409F09_scribe1.pdf

[8] Jonathan Kelner. *Properties of the Laplacian, positive semidefinite matricies, spectra of common graphs.* https://ocw.mit.edu/courses/mathematics/18-409-topics-in-theoretical-computer-science-an-algorithmists-toolkit-fall-2009/lecture-notes/MIT18_409F09_scribe2.pdf

Appendix

symm_matrix.lean

```
import data.complex.basic
import data.complex.module
import data.fintype.basic
import data.real.basic
import linear_algebra.matrix

/-!
# Symmetric Matrices

This module defines symmetric matrices, together with key properties
about their eigenvalues & eigenvectors.
It uses a more restrictive definition of eigenvalues & eigenvectors,
together with helping lemmas for vector-matrix
operations and tools for complex numbers/vectors.

## Main definitions

* `vec_conj x` - the complex conjugate of a complex vector `x`
* `vec_re x` - the vector containing the real parts of elements from x
* `vec_im x` - the vector containing the imaginary parts of elements from x

* `has_eigenpair M  x` - matrix `M` has non-zero eigenvector `x` with corresponding
    eigenvalue ``
* `has_eigenvector M x` - matrix `M` has non-zero eigenvector `x`
* `has_eigenvalue M `  - matrix `M` has eigenvalue ``

* `symm_matrix M` - `M` is a symmetric matrix

## Main statements

1. If x is an eigenvector of matrix M, then a  x is an eigenvector of M, for any
    non-zero a : .
2. If there are two eigenvectors of M that have the same correspoding eigenvalue, then
    any linear combination of them
is also an eigenvector of M with the same eigenvalue .
3. All eigenvalues of a symmetric real matrix M are real.
4. For every real eigenvalue of a symmetric matrix M, there exists a corresponding
    real-valued eigenvector.
5. If v and w are eigenvectors of a symmetric matrix M with different eigenvalues, then
    v and w are orthogonal.
-/

open_locale matrix big_operators
open fintype finset matrix complex

universes u
variables { : Type u}
variables {m n : Type*} [fintype m] [fintype n]

lemma vec_eq_unfold (x y : n  ) : ( i : n, x i) = ( i : n, y i)
  i : n, x i = y i :=
begin
 split,
 { intros hyp i, exact congr_fun hyp i },
 { intro hyp, ext, apply hyp }
end

-- ## Coercions
instance : has_coe (n  ) (n  ) :=  x, ( i, x i, 0)
instance : has_coe (matrix m n ) (matrix m n ) :=
```

```
  M, ( i j, M i j, 0)

-- ## Lemmas on

lemma conj_of_zero_im { : } (H_im : .im = 0) : conj  =  :=
by { ext; simp only [conj_re, conj_im, H_im, neg_zero] }

lemma sum_complex_re {x : n  } : ( i : n, x i).re =
  i : n, (x i).re := complex.re_lm.map_sum

lemma sum_complex_im {x : n  } : ( i : n, x i).im =
  i : n, (x i).im := complex.im_lm.map_sum

-- The real and complex parts of a complex vector
def vec_re (x : n  ) (i : n) :  := (x i).re
def vec_im (x : n  ) (i : n) :  := (x i).im

-- Defining the complex conjugate of a complex vector
section vec_conj

def vec_conj (x : n  ) (i : n) :  := conj (x i)

-- ( x)* = *  x*
lemma vec_conj_smul ( : ) (x : n  ) :
 vec_conj (  x) = (conj )  (vec_conj x) :=
by { ext ; simp only [vec_conj, algebra.id.smul_eq_mul, pi.smul_apply, ring_hom.map_mul] }

-- A i j = (A i j)
lemma coe_matrix_coe_elem (i : m) (j : n) (A : matrix m n ) : (A : matrix m n ) i j = (A i j) := rfl

-- (A  x)* = A  x*
lemma vec_conj_mul_vec [decidable_eq n] [nonempty n] (A : matrix m n ) (x : n  ) :
 vec_conj ((A : matrix m n ).mul_vec x) = (A : matrix m n ).mul_vec (vec_conj x) :=
begin
 ext,
 simp only [vec_conj, mul_vec, dot_product, conj_re, coe_matrix_coe_elem, sum_complex_re,
    mul_re, of_real_im, zero_mul],
 simp only [vec_conj, mul_vec, dot_product, conj_im, coe_matrix_coe_elem, sum_complex_im,
    mul_im, add_zero, of_real_im,
   zero_mul, sum_neg_distrib, mul_neg_eq_neg_mul_symm]
end

lemma vec_norm_sq_zero {x : n  } (H_dot : dot_product (vec_conj x) x = 0) : x = 0 :=
begin
 unfold dot_product at H_dot,
 simp only [vec_conj, mul_comm, mul_conj, complex.ext_iff, sum_complex_re, zero_re, of_real_re]
    at H_dot,
 cases H_dot with H_re H_im,
 have key :  i in (univ : finset n), norm_sq (x i) = 0  i  (univ : finset n), norm_sq(x i) = 0,
 { apply sum_eq_zero_iff_of_nonneg, intros i h_univ, exact norm_sq_nonneg (x i) },
 simp only [forall_prop_of_true, mem_univ, monoid_with_zero_hom.map_eq_zero] at key,
 rw key at H_re,
 ext i;
 { specialize H_re i, simp only [H_re, pi.zero_apply] }
end

lemma coe_vec_re (x : n  ) {i : n} : (vec_re x : n  ) i = ((x i).re : ) :=
by simpa only [vec_re]

lemma vec_add_conj_eq_two_re (x : n  ) : x + vec_conj x = (2 : )  (vec_re x : n  ) :=
begin
 ext,
 { simp [vec_conj, coe_vec_re x], linarith },
 { simp [vec_conj, coe_vec_re x] }
end
```

```
lemma vec_conj_add_zero {x : n  } (H : x + vec_conj x = 0) : vec_re x = 0 :=
begin
 rw [vec_add_conj_eq_two_re, smul_eq_zero] at H,
 cases H with H_20 H_x,
 { exfalso, simp at H_20, assumption }, -- 2 = 0
 { rw function.funext_iff at H_x,
  ext i,
  specialize H_x i,
  rw coe_vec_re at H_x,
  simp only [of_real_eq_zero, pi.zero_apply] at H_x,
  simp only [vec_re, H_x, pi.zero_apply] }
end

end vec_conj

namespace matrix

variables (M : matrix n n )

def Coe (M : matrix m n ) := (M : matrix m n )

/--
## Matrix definitions
Let `M` be a square real matrix. An `eigenvector` of `M` is a complex vector `x` with `M  x =
      x`
for some `  `, which is called the `eigenvalue` of `M` corresponding to the `eigenvector x`.
-/
def has_eigenpair ( : ) (x : n  ) : Prop :=
 x  0  (mul_vec M.Coe x =   x)

def has_eigenvector (x : n  ) : Prop :=
  : , M.has_eigenpair  x

def has_eigenvalue ( : ) : Prop :=
  x : n  , M.has_eigenpair  x

def symm_matrix : Prop := M = M

-- ## Matrix : Helping lemmas

-- (M) = M
lemma coe_transpose_matrix : (M.Coe) = (M).Coe := by { unfold Coe, ext, tidy }

-- M i j = (M i j)
lemma coe_matrix_coe_elem (i j : n) : (M.Coe) i j = (M i j) := rfl

-- (M x)* = M x*
lemma vec_conj_mul_vec_re (x : n  ) :
 vec_conj (mul_vec M.Coe x) = mul_vec (M.Coe) (vec_conj x) :=
begin
 ext ;
 simp only [vec_conj, mul_vec, dot_product, coe_matrix_coe_elem],
 { simp only [sum_complex_re, of_real_im, zero_mul, conj_re, mul_re] },
 { simp only [sum_complex_im, add_zero, of_real_im, zero_mul,
          sum_neg_distrib, conj_im, mul_neg_eq_neg_mul_symm, mul_im] },
end

lemma symm_matrix_coe (H_symm : symm_matrix M) : (M.Coe) = (M.Coe) :=
begin
 unfold symm_matrix at H_symm,
 rw [coe_transpose_matrix, H_symm]
end

-- v (M w) = (v M) w
lemma dot_product_mul_vec_vec_mul (v w : n  ) :
 dot_product v (mul_vec M.Coe w) = dot_product (vec_mul v M.Coe) w :=
```

```
begin
 have key : vec_mul v M.Coe =  j, dot_product v ( i, M.Coe i j),
 { ext ; unfold vec_mul },
 rw [key, dot_product_assoc v M.Coe w],
 ext ; simp only [dot_product, mul_vec],
end

-- 1. If x is an eigenvector of M, then a  x is an eigenvector of M, for any non-zero a :
      .
theorem has_eigenvector_smul (a : ) (x : n  ) (H_na : a  0) (H_eigenvector : has_eigenvector M x) :
 has_eigenvector M (a  x) :=
begin
 rcases H_eigenvector with , H_nx, H_mul,
 use , -- corresponding eigenvalue
 split,
 { intro hyp, rw smul_eq_zero at hyp, tauto }, -- a  x  0
 calc (M.Coe).mul_vec (a  x)
    = a  M.Coe.mul_vec x : -- M  (a  x) = a  (M  x)
 by { rw mul_vec_smul_assoc }
 ... = a  (  x) :            -- ... = a  (  x)
 by { rw H_mul }
 ... =  (a  x) :             -- ... =  (a  x)
 by { simp only [smul_smul, mul_comm] }
end

-- 2. If there are two eigenvectors that have the same correspoding eigenvalue ,
-- then any non-zero linear combination of them is also an eigenvector with the same
    eigenvalue .
theorem has_eigenpair_linear (a b : ) (v w : n  ) ( : ) (H_ne : a  v + b  w  0)
(H : has_eigenpair M  v) (H : has_eigenpair M  w) : has_eigenpair M  (a  v + b  w) :=
begin
 rcases H with H, H,
 rcases H with H, H,
 use H_ne, -- a  v + b  w  0
 calc M.Coe.mul_vec (a  v + b  w) -- M  (a  v + b  w) = M  (a  v) + M  (b  w)
    = M.Coe.mul_vec(a  v) + M.Coe.mul_vec(b  w) :
 by { ext ; simp only [mul_vec, pi.add_apply, dot_product_add] }
 ... = a  M.Coe.mul_vec v + b  M.Coe.mul_vec w :  -- ... = a  (M  v) + b  (M  w)
 by { ext ; simp only [mul_vec, algebra.id.smul_eq_mul,
                dot_product_smul, pi.add_apply, pi.smul_apply] }
 ... = a  (  v) + b  (  w) :              -- ... = a  (  v) + b  (  w)
 by { rw [H, H] }
 ... =  (a  v + b  w) :                   -- ... =  (a  v + b  w)
 by { simp only [smul_smul, mul_comm, smul_add] }
end

-- 3. All eigenvalues of a symmetric real matrix M are real.
theorem symm_matrix_real_eigenvalues (H_symm : symm_matrix M) :
  ( : ), has_eigenvalue M  .im = 0 :=
begin
 -- (1) M  x =  x
 rintro  x, H_x, H_eq,
 -- (2) M  x* = *  x*
 have H_eq : mul_vec M.Coe (vec_conj x) = (conj )  (vec_conj x),
 { rw [vec_conj_smul  x, M.vec_conj_mul_vec_re x, H_eq] },
 -- (3)  ((x*) x) = * ((x*) x)
 have H_eq :  * (dot_product (vec_conj x) x) = conj  * dot_product (vec_conj x) x,

 calc  * dot_product (vec_conj x) x
    = dot_product (vec_conj x) (  x) :   --  ((x*) x) = (x*) (  x)
 by { rw smul_dot_product  (vec_conj x) x,
     simp [dot_product, vec_conj, mul_assoc, mul_comm] }
 ... = dot_product (vec_conj x) (M.Coe.mul_vec x) : -- ... = (x*) (M x)
 by { rw H_eq }
 ... = dot_product (M.Coe.vec_mul (vec_conj x)) x : -- ... = ((x*) M) x
 by { exact (dot_product_assoc (vec_conj x) M.Coe x).symm }
```

```
  ... = dot_product (M.Coe.mul_vec (vec_conj x)) x : -- ... = (M x*) x
  by { rw mul_vec_transpose M.Coe (vec_conj x) }
  ... = dot_product (M.Coe.mul_vec (vec_conj x)) x :  -- ... = (M x*) x
  by { have H : M.Coe = M.Coe, { unfold Coe, tidy }, rw H }
  ... = dot_product (conj   vec_conj x) x :        -- ... = (* x*) x
  by { rw H_eq }
  ... = conj  * dot_product (vec_conj x) x :        -- ... = * ((x*) x)
  by { rw smul_dot_product (conj ) (vec_conj x) x },

  -- (4) ( - *) ((x*) x) = 0
  have H_eq : ( − conj ) * dot_product (vec_conj x) x = 0,
  { rw sub_mul, simp only [H_eq, sub_self] },
  --  - * = 0  (x*) x = 0
  rw mul_eq_zero at H_eq,
  cases H_eq with H_ H_prod,
  { rw [sub_eq_zero, eq_comm, eq_conj_iff_real] at H_,
    cases H_ with r H_r,
    rw [H_r, of_real_im] }, --  - * = 0
  { exfalso,
    exact H_x (vec_norm_sq_zero H_prod) }, -- (x*) x = 0
end

-- 4. For every real eigenvalue of a symmetric matrix M, there exists a corresponding
     real-valued eigenvector.
theorem symm_matrix_real_eigenvectors (H_symm : symm_matrix M) ( : ) (H_eigenvalue :
    has_eigenvalue M ) :
  x : n  , has_eigenpair M  x  vec_im x = 0 :=
begin
  -- We know that    from before.
  have H_ : .im = 0,
  { apply M.symm_matrix_real_eigenvalues H_symm  H_eigenvalue },
  rcases H_eigenvalue with x, H_nx, H_mul,
  by_cases H_re : vec_re x = 0,
  -- 1) I  x will be used
  { use (I  x),
    split,
    -- 1.1) I  x is an eigenvector
    { split,
      -- 1.1.1) I  x  0
      { intro hyp, rw smul_eq_zero at hyp,
        have H_nI : I  0, { exact I_ne_zero },
        tauto },
      -- 1.1.2) M (I  x) =   (I  x)
      { simp only [mul_vec_smul_assoc, H_mul, smul_smul, mul_comm] } },
    -- 1.2) I  x
    { ext i,
      simp only [vec_re, vec_eq_unfold] at H_re,
      simp only [vec_im, algebra.id.smul_eq_mul, I_re, one_mul,
             I_im, zero_mul, mul_im, zero_add, pi.smul_apply],
      exact H_re i } },
  -- 2) x + x* will be used
  { use (x + vec_conj x),
    split,
    -- 2.1) x + x* is an eigenvector
    { split,
      -- 2.1.1) x + x*  0
      { intro hyp, exact H_re (vec_conj_add_zero hyp) },
      -- 2.1.2) M (x + x*) =  (x + x*)
      { calc M.Coe.mul_vec (x + vec_conj x)
          = M.Coe.mul_vec x + M.Coe.mul_vec (vec_conj x) :
        by { apply mul_vec_add } -- M (x + x*) = M x + M x*
        ... = M.Coe.mul_vec x + vec_conj (M.Coe.mul_vec x) :
        by { rw M.vec_conj_mul_vec_re x }    -- ... = M x + (M x)*
        ... =  x + vec_conj ( x) :
        by { rw H_mul }                  -- ... =  x + ( x)*
        ... =  x + (conj ) (vec_conj x) :
```

```
      by { rw vec_conj_smul }              -- ... =   x + *  x*
      ... =  x +  (vec_conj x) :
      by { rw conj_of_zero_im H_ }         -- ... =   x +   x*
      ... =  (x + vec_conj x) :
      by { simp only [smul_add] } } },     -- ... =   (x + x*)
   -- 2.2) x + x*
   { ext, simp [vec_add_conj_eq_two_re, vec_im, coe_vec_re] } }
end

-- 5. If v and w are eigenvectors of a symmetric matrix M with different eigenvalues,
    then v and w are orthogonal.
theorem dot_product_neq_eigenvalue_zero (H_symm : symm_matrix M) (v w : n  ) ( ' : )
(H_ne :  ') (H : has_eigenpair M  v) (H : has_eigenpair M ' w) : dot_product v w = 0 :=
begin
 have key : ( - ') * dot_product v w = 0,
 calc ( - ') * dot_product v w
    =  * dot_product v w - ' * dot_product v w :
 by { apply mul_sub_right_distrib }-- ( - ')v w = (v w) - '(v w)
 ... = dot_product (  v) w - dot_product v ('  w) :
 by { simp only [dot_product_smul,
    smul_dot_product] }                -- ... = (  v)w - v('   w)
 ... = dot_product (M.Coe.mul_vec v) w - dot_product v (M.Coe.mul_vec w) :
 by { rw [H.2, H.2] }                  -- ... = (M v)w - v(M w)
 ... = dot_product (M.Coe.mul_vec v) w - dot_product (vec_mul v M.Coe) w :
 by { rw M.dot_product_mul_vec_vec_mul v w }-- ... = (M v)w - (v M)w
 ... = dot_product (M.Coe.mul_vec v) w - dot_product (vec_mul v M.Coe) w:
 by { rw symm_matrix_coe M H_symm }  -- ... = (M v)w - (v M)w
 ... = dot_product (M.Coe.mul_vec v) w - dot_product (mul_vec M.Coe v) w :
 by { rw vec_mul_transpose M.Coe v }      -- ... = (M v)w - (M v)w
 ... = 0 :
 by { simp only [sub_self] },           -- ... = 0
 rw mul_eq_zero at key,
 cases key with H_ H_dot,
 { exfalso, rw sub_eq_zero at H_, exact H_ne H_ }, --  - ' = 0
 { exact H_dot } -- v w = 0
end

end matrix
```

incidence.lean

```
import algebra.big_operators.basic
import combinatorics.simple_graph.basic
import data.fintype.basic
import data.sym2
import linear_algebra.matrix

/-!
# Incidence matrices

This module defines the incidence matrix `inc_matrix` of an undirected graph `
    simple_graph`, and provides
theorems and lemmas connecting graph properties to computational properties of the
    matrix. It also
defines the notion of `orientation` for a `simple_graph`, picking a direction for each
    undirected
edge in the graph and then defining the oriented incidence matrix `oriented_inc_matrix`
    based on that.

## Main definitions

* `inc_matrix` is the incidence matrix `M` of a `simple_graph` with coefficients in a given
    ring R.
* `orientation` is a structure that defines a choice of direction on the edges of a `
    simple_graph`.
* `oriented_inc_matrix` is the oriented incidence matrix `N(o)` of a `simple_graph` with
respect to a given `orientation`.

## Main statements

1.  e : E, M i e * M j e = 1, for any two adjacent vertices i and j.
2. M i e * M j e = 0, for any two distinct non-adjacent vertices i, j and edge e.
3. Every element from M is idempotent.
4. For any vertex i, the sum on the ith row of M is equal to the degree of i.
5. (N(o) i e) ^ 2 = M i e, for any orientation o, vertex i and edge e.
6. For any adjacent vertices i j and edge e, N(o) i e * N(o) j e = if e = (i, j) then -1
    else 0.
7. For any non-adjacent distinct vertices i j and edge e, N(o) i e * N(o) j e = 0.
8. (x  N(o)) e = x o.head(e) - x o.tail(e).
-/

open_locale big_operators matrix
open finset matrix simple_graph sym2

universe u
variables {R : Type u} [ring R] [nontrivial R] [decidable_eq R]

@[simp]
lemma ite_prod_one_zero {P Q : Prop} [decidable P] [decidable Q] :
  (ite P (1 : R) 0) * (ite Q 1 0) = ite (P  Q) 1 0 :=
by { by_cases h : P; simp [h] }

lemma fintype.card_coe_filter { : Sort*} {s t : set } [fintype s] [fintype t]
  [decidable_pred ( (x : t), (x : )  s)] (h : s  t) :
  fintype.card s = finset.card (finset.filter ( (x : t), (x : )  s) finset.univ) :=
begin
  refine finset.card_congr _ _ _ _,
  { rintros e, he he′,
    exact e, h he },
  { rintros e, he he′,
    simpa only [true_and, finset.mem_univ, finset.mem_filter] using he},
  { rintros e1, he1 e2, he2 he1′ he2′ hr,
    ext,
    simp only [subtype.mk_eq_mk] at hr,
```

```
   simp only [hr] },
  { rintros e, he he',
   use [e],
   { simpa only [true_and, finset.mem_univ, finset.mem_filter] using he'},
   { simp only [finset.mem_univ, exists_prop_of_true] } }
end

namespace simple_graph

universe v
variables {V : Type v} [fintype V] (G : simple_graph V) (R) [decidable_rel G.adj] [decidable_eq V]

-- ## Incidence matrix M

/-- `inc_matrix G R` is the matrix `M` such that `M i e = 1` if vertex `i` is an
endpoint of the edge `e` in the simple graph `G`, otherwise `M i j = 0`. -/
def inc_matrix : matrix V G.edge_set R
| i e := if (e : sym2 V)  G.incidence_set i then 1 else 0

@[simp]
lemma inc_matrix_apply {i : V} {e : G.edge_set} :
 G.inc_matrix R i e = if (e : sym2 V)  G.incidence_set i then 1 else 0 := rfl

lemma inc_matrix_def : G.inc_matrix R =  i e, ite ((e : sym2 V)  G.incidence_set i) 1 0 :=
by { ext, simp only [inc_matrix_apply] }

-- ### Relation between inc_matrix elements and incidence_set property

@[simp]
lemma inc_matrix_zero {i : V} {e : G.edge_set} : G.inc_matrix R i e = 0  e.val  G.incidence_set i :
    =
by simp only [inc_matrix, ite_eq_right_iff, subtype.val_eq_coe, decidable.not_imp_not,
         forall_true_left, not_false_iff, one_ne_zero]

@[simp]
lemma inc_matrix_one {i : V} {e : G.edge_set} : G.inc_matrix R i e = 1  e.val  G.incidence_set i :=
by simp only [inc_matrix, ite_eq_left_iff, subtype.val_eq_coe, decidable.not_imp_not,
         set.not_not_mem, forall_true_left, not_false_iff, zero_ne_one]

-- ### One - zero properties

@[simp]
lemma inc_matrix_not_zero {i : V} {e : G.edge_set} : ň G.inc_matrix R i e = 0  G.inc_matrix R i e
    = 1 :=
by simp only [inc_matrix_zero, inc_matrix_one, set.not_not_mem]

@[simp]
lemma inc_matrix_not_one {i : V} {e : G.edge_set} : ň G.inc_matrix R i e = 1  G.inc_matrix R i e =
    0 :=
by simp only [inc_matrix_zero, inc_matrix_one]

lemma inc_matrix_zero_or_one {i : V} {e : G.edge_set} :
 G.inc_matrix R i e = 0  G.inc_matrix R i e = 1 :=
by { rw [inc_matrix_zero, inc_matrix_one], exact (em (e.val  G.incidence_set i)).symm }

@[simp]
lemma inc_matrix_elements_product_one {i j : V} {e : G.edge_set} :
 G.inc_matrix R i e * G.inc_matrix R j e = 1  G.inc_matrix R i e = 1  G.inc_matrix R j e = 1 :=
begin
 cases G.inc_matrix_zero_or_one R with H H,
 { rw H, simp only [if_t_t, mul_boole, inc_matrix_apply, zero_ne_one, false_and] },
 { rw H, simp only [true_and, mul_boole, inc_matrix_apply, eq_self_iff_true] }
end

-- ### Helping lemmas for edges
```

```
@[simp]
lemma edge_val_equiv {e e : G.edge_set} : e.val = e.val  e = e :=
begin
 split,
 { exact subtype.eq },
 { intro hyp,
  rw hyp }
end

lemma edge_val_in_set {e : G.edge_set} : e.val  G.edge_set :=
by simp only [subtype.coe_prop, subtype.val_eq_coe]

lemma edge_set_ne {u v : V} {e: G.edge_set} (h : e.val = (u, v)) : u  v :=
begin
 apply G.ne_of_adj,
 simp only [G.mem_edge_set, h, edge_val_in_set],
end

lemma incidence_equiv {i : V} {e : G.edge_set} : e.val  G.incidence_set i  i  e.val :=
by simp only [incidence_set, true_and, set.mem_sep_eq, edge_val_in_set]

lemma incidence_set_iff_any_vertex {i u v : V} (h : (u, v)  G.edge_set) :
 (u, v)  G.incidence_set i  i = u  i = v :=
by simp only [mem_iff, h, incidence_set, true_and, set.mem_sep_eq]

lemma edge_in_two_incidence_sets {i j : V} {e : sym2 V} (H_ne : i  j) :
 e  G.incidence_set i  e  G.incidence_set j  e = (i, j) :=
begin
 refine quotient.rec_on_subsingleton e ( p, _),
 rcases p with v, w,
 rw eq_iff,
 rintros _, H_i, _, H_j,
 cases (mem_iff.mp H_i) with H_i H_i;
 cases (mem_iff.mp H_j) with H_j H_j,
 { exfalso, apply H_ne, rw [H_i, H_j] }, -- i = v, j = v
 { left, use [H_i.symm, H_j.symm] },     -- i = v, j = w
 { right, use [H_j.symm, H_i.symm] },    -- i = w, j = v
 { exfalso, apply H_ne, rw [H_i, H_j] }  -- i = w, j = w
end

lemma mem_incidence_sets_iff_eq {i j : V} {e : sym2 V} (h : G.adj i j) :
 e  G.incidence_set i  e  G.incidence_set j  e = (i, j) :=
begin
 refine quotient.rec_on_subsingleton e ( p, _),
 rcases p with v, w,
 rw eq_iff,
 simp only [incidence_set],
 tidy,
end

lemma adj_iff_exists_edge_val {i j : V} : G.adj i j   (e : G.edge_set), e.val = (i, j) :=
by simp only [mem_edge_set, exists_prop, set_coe.exists, exists_eq_right, subtype.coe_mk]

-- 1.  e : E, M i e * M j e = 1, where i and j are adjacent.
theorem adj_sum_of_prod_inc_one {i j : V} (H_adj : G.adj i j) :
  (e : G.edge_set), G.inc_matrix R i e * G.inc_matrix R j e = (1 : R) :=
begin
 simp only [inc_matrix_apply, ite_prod_one_zero, sum_boole,
         G.mem_incidence_sets_iff_eq H_adj, subtype.val_eq_coe],
 rw adj_iff_exists_edge_val at H_adj,
 rcases H_adj with e, H_e,
 simp only [H_e, edge_val_equiv],
 have H : filter ( (x : G.edge_set), x = e) univ = {e},
 { ext, simp only [true_and, mem_filter, mem_univ, mem_singleton] },
 simp only [H, filter_congr_decidable, nat.cast_one, card_singleton]
end
```

```
-- 2. M i e * M j e = 0, where i, j distinct non-adjacent vertices, e an edge.
theorem inc_matrix_prod_non_adj {i j : V} {e : G.edge_set} (Hne : i  j) (H_non_adj : ¬ G.adj i j) :
 G.inc_matrix R i e * G.inc_matrix R j e = 0 :=
begin
 by_cases H : G.inc_matrix R i e = 0,
 { rw [H, zero_mul] },
 { rw [inc_matrix_not_zero, inc_matrix_one] at H,
  by_cases H : G.inc_matrix R j e = 0,
  { rw [H, mul_zero] },
  { rw [inc_matrix_not_zero, inc_matrix_one] at H,
   exfalso,
   apply H_non_adj,
   rw [mem_edge_set, G.edge_in_two_incidence_sets Hne H, H],
   exact G.edge_val_in_set } }
end

-- 3. (M i e) ^ 2 = M i e; with i a vertex, e an edge.
@[simp]
theorem inc_matrix_element_power_id {i : V} {e : G.edge_set} :
 (G.inc_matrix R i e) * (G.inc_matrix R i e) = G.inc_matrix R i e :=
by simp [inc_matrix_apply]

-- 4. degree(i) =  e : E, M i e; where i is a vertex.
theorem degree_equals_sum_of_incidence_row {i : V} :
 (G.degree i : R) =  (e : G.edge_set), G.inc_matrix R i e :=
begin
 rw [inc_matrix_def, card_incidence_set_eq_degree],
 simp only [sum_boole, nat.cast_inj, fintype.card_coe_filter (G.incidence_set_subset i)],
end

-- ## Orientations

/-- Define an `orientation` on the undirected graph G as a structure that defines
    (consistently)
for each edge a `head` and a `tail`. -/
@[ext]
structure orientation (G : simple_graph V) :=
(head : G.edge_set  V)
(tail : G.edge_set  V)
(consistent (e : G.edge_set) : e.val = (head(e),tail(e)))

-- ## Oriented Incidence Matrix N(o)

/-- An `oriented incidence matrix` N(o) is defined with respect to the orientation of the
    edges and is defined to be
`1` for entries (`i`,`e`) where `i` is the head of `e`, `-1` where `i` is the tail of `e`, and `
    0` otherwise. -/
def oriented_inc_matrix (o : orientation G) : matrix V G.edge_set R :=
 i e, if i = o.head e then (1 : R) else (if i = o.tail e then -1 else 0)

variables {o : orientation G}

@[simp]
lemma oriented_inc_matrix_apply {i : V} {e : G.edge_set} :
 G.oriented_inc_matrix R o i e = if i = o.head e then 1 else (if i = o.tail e then (-1 : R) else
    0) := rfl

lemma head_neq_tail {e : G.edge_set} : o.head(e)  o.tail(e) := G.edge_set_ne (o.consistent e)

@[simp]
lemma oriented_inc_matrix_head {i : V} {e : G.edge_set} (H_head : i = o.head e) :
 G.oriented_inc_matrix R o i e = 1 :=
by simp only [H_head, if_true, eq_self_iff_true, oriented_inc_matrix_apply]

@[simp]
```

```
lemma oriented_inc_matrix_tail {i : V} {e : G.edge_set} (H_tail : i = o.tail e) :
 G.oriented_inc_matrix R o i e = -1 :=
by simp only [H_tail, oriented_inc_matrix, (G.head_neq_tail).symm, if_false, if_true,
    eq_self_iff_true]

@[simp]
lemma oriented_inc_matrix_zero {i : V} {e : G.edge_set} :
 G.oriented_inc_matrix R o i e = 0  i  o.head e  i  o.tail e :=
begin
 by_cases H : i = o.head e,
 { simp only [oriented_inc_matrix, H, if_true, eq_self_iff_true, not_true,
         ne.def, one_ne_zero, false_and] },
 { by_cases H : i = o.tail e,
  { simp only [H, oriented_inc_matrix_tail, eq_self_iff_true, not_true,
          ne.def, neg_eq_zero, one_ne_zero, and_false] },
  { simp only [H, H, eq_self_iff_true, if_false, ne.def,
          not_false_iff, and_self, oriented_inc_matrix_apply] } }
end

@[simp]
lemma oriented_inc_matrix_non_zero {i : V} {e : G.edge_set} :
 ň G.oriented_inc_matrix R o i e = 0  i = o.head e  i = o.tail e :=
begin
 by_cases H : i = o.head e,
 { simp only [H, if_true, true_or, eq_self_iff_true, ne.def,
         not_false_iff, one_ne_zero, oriented_inc_matrix_apply] },
 { by_cases H : i = o.tail e,
  { simp only [H, oriented_inc_matrix_tail, eq_self_iff_true, ne.def, or_true,
          not_false_iff, neg_eq_zero, one_ne_zero] },
  { simp only [H, H, eq_self_iff_true, not_true, if_false,
          ne.def, oriented_inc_matrix_apply, or_self] } }
end

lemma incidence_set_orientation_head {e : G.edge_set} : e.val  G.incidence_set (o.head e) :=
by { rw [incidence_equiv, o.consistent e], simp only [mem_iff, true_or, eq_self_iff_true] }

lemma incidence_set_orientation_tail {e : G.edge_set} : e.val  G.incidence_set (o.tail e) :=
by { rw [incidence_equiv, o.consistent e], simp only [mem_iff, eq_self_iff_true, or_true] }

lemma incidence_set_orientation {i : V} {e : G.edge_set} :
 e.val  G.incidence_set i  i = o.head e  i = o.tail e :=
begin
 rw o.consistent e,
 have key : (o.head e, o.tail e)  G.edge_set, {rw o.consistent e, exact G.edge_val_in_set},
 exact G.incidence_set_iff_any_vertex key,
end

lemma not_inc_set_orientation {i : V} {e : G.edge_set}
 (H_head : i  o.head e) (H_tail : i  o.tail e) : e.val  G.incidence_set i :=
begin
 intro h,
 rw G.incidence_set_orientation at h,
 tauto,
end

-- 5. (N(o) i e) ^ 2 = M i e, for any orientation o, vertex i and edge e.
@[simp]
theorem oriented_inc_matrix_elem_squared {i : V} {e : G.edge_set} :
 G.oriented_inc_matrix R o i e * G.oriented_inc_matrix R o i e = G.inc_matrix R i e :=
begin
 by_cases H_head : i = o.head e,
 { rw [G.oriented_inc_matrix_head R H_head, H_head, mul_one, eq_comm, inc_matrix_one],
  exact G.incidence_set_orientation_head },
 { by_cases H_tail : i = o.tail e,
  { rw [G.oriented_inc_matrix_tail R H_tail, H_tail, mul_neg_eq_neg_mul_symm, mul_one,
      neg_neg, eq_comm, inc_matrix_one],
```

```
    exact G.incidence_set_orientation_tail },
  { rw [(G.oriented_inc_matrix_zero R).mpr H_head, H_tail, mul_zero, eq_comm, inc_matrix_zero],
    exact G.not_inc_set_orientation H_head H_tail } }
end

-- 6. For any adjacent vertices i j and edge e, N(o) i e * N(o) j e = if e = (i, j) then
    -1 else 0.
theorem oriented_inc_matrix_prod_of_adj {i j : V} {e : G.edge_set} (H_adj : G.adj i j) :
 G.oriented_inc_matrix R o i e * G.oriented_inc_matrix R o j e = ite (e.val = (i, j)) (-1) 0 :=
begin
 by_cases H_e : e.val = (i, j),
 -- 1) e is the edge between i and j
 { rw [H_e, if_pos rfl],
  rw [o.consistent e, eq_iff] at H_e,
  rcases H_e with (H_head_i, H_tail_j | H_head_j, H_tail_i),
  { rw [G.oriented_inc_matrix_head R H_head_i.symm, G.oriented_inc_matrix_tail R
    H_tail_j.symm,
      mul_neg_eq_neg_mul_symm, mul_one] },
  { rw [G.oriented_inc_matrix_head R H_head_j.symm, G.oriented_inc_matrix_tail R
    H_tail_i.symm, mul_one] } },
 -- 2) e is not the edge between i and j
 { simp only [H_e, if_false],
  rw [o.consistent e, eq_iff, decidable.not_or_iff_and_not] at H_e,
  repeat { rw decidable.not_and_iff_or_not at H_e },
  rcases H_e with (H_head_i | H_tail_j), (H_head_j | H_tail_i),
  -- 2.1) both i and j are not the head of e
  { have H_tail : o.tail e  i  o.tail e  j,
   { by_contradiction h,
    rw [decidable.not_or_iff_and_not, not_not, not_not] at h,
    rcases h with h_i, h_j, rw h_i at h_j,
    exact G.ne_of_adj H_adj h_j },
   cases H_tail with H_tail_i H_tail_j,
   -- 2.1.1) i is not the tail of e
   { rw [(G.oriented_inc_matrix_zero R).mpr ne.symm H_head_i, ne.symm H_tail_i, zero_mul] },
   -- 2.1.2) j is not the tail of e
   { rw [(G.oriented_inc_matrix_zero R).mpr ne.symm H_head_j, ne.symm H_tail_j, mul_zero] } },
  -- 2.2) i is neither the head of e nor its tail
  { rw [(G.oriented_inc_matrix_zero R).mpr ne.symm H_head_i, ne.symm H_tail_i, zero_mul] },
  -- 2.3) j is neither the head of e nor its tail
  { rw [(G.oriented_inc_matrix_zero R).mpr ne.symm H_head_j, ne.symm H_tail_j, mul_zero] },
  -- 2.4) both i and j are not the tail of e
  { have H_head : o.head e  i  o.head e  j,
   { by_contradiction h,
    rw [decidable.not_or_iff_and_not, not_not, not_not] at h,
    rcases h with h_i, h_j, rw h_i at h_j,
    exact G.ne_of_adj H_adj h_j },
   cases H_head with H_head_i H_head_j,
   -- 2.4.1) i is not the head of e
   { rw [(G.oriented_inc_matrix_zero R).mpr ne.symm H_head_i, ne.symm H_tail_i, zero_mul] },
   -- 2.4.2) j is not the head of e
   { rw [(G.oriented_inc_matrix_zero R).mpr ne.symm H_head_j, ne.symm H_tail_j, mul_zero] } } }
end

-- 7. For any non-adjacent distinct vertices i j and edge e, N(o) i e * N(o) j e = 0.
theorem oriented_inc_matrix_prod_non_adj {i j : V} {e : G.edge_set} (H_ij : i  j) (H_not_adj : ň
    G.adj i j) :
 G.oriented_inc_matrix R o i e * G.oriented_inc_matrix R o j e = 0 :=
begin
 by_cases H : G.oriented_inc_matrix R o i e = 0,
 { rw [H, zero_mul] },
 { by_cases H : G.oriented_inc_matrix R o j e = 0,
  { rw [H, mul_zero] },
  { rcases ((G.oriented_inc_matrix_non_zero R).mp H) with (H_head_i | H_tail_i) ;
   rcases ((G.oriented_inc_matrix_non_zero R).mp H) with (H_head_j | H_tail_j),
   { rw [H_head_i, H_head_j] at H_ij, tauto },
   { exfalso, apply H_not_adj,
```

```
      rw [H_head_i, H_tail_j, mem_edge_set, o.consistent e],
      simp only [subtype.coe_prop, subtype.val_eq_coe] },
    { exfalso, apply H_not_adj, apply (G.edge_symm i j).mpr,
      rw [H_tail_i, H_head_j, mem_edge_set, o.consistent e],
      simp only [subtype.coe_prop, subtype.val_eq_coe] },
    { rw [H_tail_i, H_tail_j] at H_ij, tauto } } }
end

-- 8. (x  N(o)) e = x o.head(e) - x o.tail(e).
theorem vec_mul_oriented_inc_matrix {o : orientation G} (x : V  R) (e : G.edge_set) :
 vec_mul x (G.oriented_inc_matrix R o) e = x (o.head e) - x (o.tail e) :=
begin
 simp only [vec_mul, dot_product, oriented_inc_matrix, mul_ite, mul_one,
    mul_neg_eq_neg_mul_symm, mul_zero],
 rw [sum_ite, sum_ite, sum_filter, sum_ite_eq', sum_const_zero, add_zero, filter_filter],
 simp only [mem_univ, if_true],
 have key : filter ( (a : V), ňa = o.head e  a = o.tail e) univ = {o.tail e},
 { ext,
  simp only [mem_filter, mem_singleton, true_and, and_iff_right_iff_imp, mem_univ],
  rintro rfl,
  exact ne.symm (G.head_neq_tail) },
 rw [key, sum_singleton],
 ring_nf
end

end simple_graph
```

laplace.lean

```
import algebra.big_operators.basic
import combinatorics.simple_graph.adj_matrix
import combinatorics.simple_graph.basic
import linear_algebra.matrix
import project.incidence

/-!
# Laplacian matrices

This module defines the Laplacian matrix `laplace_matrix` of an undirected graph `
    simple_graph` and
provides theorems and lemmas connecting graph properties to computational properties of
    the matrix.

## Main definitions

* `laplace_matrix` is the Laplace matrix of a `simple_graph` with coefficients in a ring R
* `signless_laplace_matrix` is the signless Laplace matrix of a `simple_graph` with
coefficients in a ring R
* `edge_from_verts` is the edge that is created by two adjacent vertices

## Main statements

1. The degree of a vertex v is equal to the sum of elements from row v of the adjacency
    matrix.
2. The Laplacian matrix is symmetric.
3. The sum of elements on any row of the Laplacian is zero.
4. The Laplacian matrix decomposition.
5. The Laplacian is a quadratic form : x  L  x =  e : G.edge_set, (x head(e) - x
    tail(e)) ^ 2.
6. The signless Laplacian matrix decomposition.
-/

open_locale big_operators matrix
open finset matrix simple_graph

universes u v
variables {V : Type u} [fintype V] [decidable_eq V]
variables {R : Type v} [comm_ring R] [nontrivial R] [decidable_eq R]

namespace simple_graph

variables (G : simple_graph V) (R) [decidable_rel G.adj]

lemma adj_matrix_eq {i j : V} (H_eq : i = j) : G.adj_matrix R i j = 0 :=
by simp only [H_eq, irrefl, if_false, adj_matrix_apply]

lemma adj_matrix_adj {i j : V} (H_adj : G.adj i j) : G.adj_matrix R i j = 1 :=
by simp only [H_adj, adj_matrix_apply, if_true]

lemma adj_matrix_not_adj {i j : V} (H_not_adj : ň G.adj i j) : G.adj_matrix R i j = 0 :=
by simp only [H_not_adj, adj_matrix_apply, if_false]

-- 1. The degree of a vertex v is equal to the sum of elements from row v of the
    adjacency matrix.
theorem degree_eq_sum_of_adj_matrix_row { : Type*} [semiring ] {i : V} :
  (G.degree i : ) =  (j : V), G.adj_matrix  i j :=
by { rw [mul_one (G.degree i : )],
    simp only [adj_matrix_mul_vec_const_apply, mul_vec,
           dot_product, boole_mul, adj_matrix_apply] }

-- ## Laplacian matrix L

/-- `laplace_matrix G` is the matrix `L` of an `simple graph G` with ` i j  V` :
```

```
` / L i j = G.degree i`, if `i = j`
` / L i j = - A i j`, otherwise. -/
def laplace_matrix : matrix V V R
| i j := if i = j then G.degree i else - G.adj_matrix R i j

@[simp]
lemma laplace_matrix_apply {i j : V} :
 G.laplace_matrix R i j = ite (i = j) (G.degree i) (- G.adj_matrix R i j) := rfl

lemma laplace_matrix_eq {i j : V} (H_eq : i = j) : G.laplace_matrix R i j = G.degree i :=
by { rw [laplace_matrix_apply, adj_matrix_apply], simp only [H_eq, if_true, eq_self_iff_true] }

lemma laplace_matrix_neq {i j : V} (H_neq : i  j) :
 G.laplace_matrix R i j = - G.adj_matrix R i j :=
by simp only [laplace_matrix_apply, adj_matrix_apply, H_neq, if_false]

-- 2. The Laplacian matrix is symmetric.
@[simp]
theorem transpose_laplace_matrix : (G.laplace_matrix R) = G.laplace_matrix R :=
begin
 ext i j,
 by_cases H : (i = j),
 { simp only [H, transpose_apply] },
 { rw [transpose_apply, G.laplace_matrix_neq R H, G.laplace_matrix_neq R (ne.symm H)],
  simp [edge_symm] }
end

lemma filter_eq_neq_empty {i : V} [decidable_eq V] : filter (eq i) (univ \ {i}) =  :=
by { ext, tidy }

lemma filter_id {i : V} : filter ( (x : V), ǐ = x) (univ \ {i}) = (univ \ {i}) :=
by { ext, tidy }

-- 3. The sum of elements on any row of the Laplacian is zero.
theorem sum_of_laplace_row_equals_zero {i : V} :  (j : V), G.laplace_matrix R i j = 0 :=
begin
 rw [sum_eq_add_sum_diff_singleton (mem_univ i), laplace_matrix_eq],
 simp only [laplace_matrix_apply, sum_ite, filter_eq_neq_empty, filter_id, adj_matrix_apply],
 rw [sum_neg_distrib, sum_boole, sum_const, card_empty, zero_smul,
    zero_add, degree_eq_sum_of_adj_matrix_row],
 have H : filter ( (x : V), G.adj i x) (univ \ {i}) = filter (G.adj i) univ,
 { ext,
  simp only [true_and, mem_filter, mem_sdiff, and_iff_right_iff_imp, mem_univ, mem_singleton],
  intro hyp,
  exact ne.symm (G.ne_of_adj hyp) },
 simp only [H, adj_matrix_apply, sum_boole, add_right_neg, eq_self_iff_true]
end

-- L = D - A. (D = degree matrix of G)
lemma laplace_eq_degree_minus_adj :
 G.laplace_matrix R = diagonal( v, G.degree v) - G.adj_matrix R :=
begin
 ext,
 by_cases H : (i = j),
 { rw [G.laplace_matrix_eq R H, dmatrix.sub_apply, G.adj_matrix_eq R H,
     sub_zero, H, diagonal_apply_eq] },
 { rw [G.laplace_matrix_neq R H, dmatrix.sub_apply, diagonal_apply_ne H, zero_sub] }
end

def edge_from_verts (i j : V) (H_adj : G.adj i j) : G.edge_set :=
 (i, j), G.mem_edge_set.mpr H_adj

@[simp]
lemma edge_from_verts_iff {i j : V} {e : G.edge_set} (H_adj : G.adj i j) :
  e = G.edge_from_verts i j H_adj  e.val = (i, j) :=
begin
```

```
 split,
 { intro hyp, simp only [edge_from_verts, hyp] },
 { intro hyp, tidy }
end

-- 4. L = N(o)  N(o), for any orientation o.
theorem laplace_decomposition (o : orientation G) :
 G.laplace_matrix R = G.oriented_inc_matrix R o  (G.oriented_inc_matrix R o) :=
begin
 ext i j,
 by_cases H_ij : i = j,
 { rw [G.laplace_matrix_eq R H_ij, mul_apply, H_ij, G.degree_equals_sum_of_incidence_row R],
  simp only [transpose_apply, G.oriented_inc_matrix_elem_squared R] },
 { rw [G.laplace_matrix_neq R H_ij, mul_apply],
  by_cases H_adj : G.adj i j,
  { simp only [G.adj_matrix_adj R H_adj, transpose_apply,
          G.oriented_inc_matrix_prod_of_adj R H_adj],
   have key :  (e : G.edge_set),
    ite (e.val = (i, j)) (-1 : R) 0 = - ite (e.val = (i, j)) 1 0,
   { intro e,
    convert (apply_ite ( x : R, -x) (e.val = (i, j)) 1 0).symm,
    rw neg_zero },
   have sum :  (e : G.edge_set), ite (e.val = (i, j)) (-1 : R) 0 =
          (e : G.edge_set), - ite (e.val = (i, j)) (1 : R) 0,
   { simp only [key] },
   rw [sum, sum_hom, neg_inj, sum_boole],
   have key : filter ( (e : G.edge_set), e.val = (i, j)) univ =
          {G.edge_from_verts i j H_adj},
   { ext,
    simp only [true_and, mem_filter, mem_univ, mem_singleton],
    rw G.edge_from_verts_iff H_adj },
   rw key,
   simp only [nat.cast_one, card_singleton] },
  { simp only [G.adj_matrix_not_adj R H_adj, transpose_apply, sum_const_zero,
          G.oriented_inc_matrix_prod_non_adj R H_ij H_adj, neg_zero] } }
end

-- 5. The Laplacian is a quadratic form : x L x =  e : G.edge_set, (x head(e) - x
    tail(e))  ^ 2.
theorem laplace_quadratic_form {o : orientation G} (x : V  R) :
 dot_product (vec_mul x (G.laplace_matrix R)) x =
  e : G.edge_set, (x (o.head e) - x (o.tail e)) ^ 2 :=
by calc dot_product (vec_mul x (G.laplace_matrix R)) x
     = dot_product (vec_mul x (G.oriented_inc_matrix R o  (G.oriented_inc_matrix R o))) x :
  by { rw laplace_decomposition } -- x  L  x = x  (N(o)  N(o))  x
  ... = dot_product (vec_mul (vec_mul x (G.oriented_inc_matrix R o))
              (G.oriented_inc_matrix R o)) x :
  by { rw vec_mul_vec_mul }          -- ... = (x  N(o))  N(o)  x
  ... = dot_product ( j, dot_product (vec_mul x (G.oriented_inc_matrix R o))
              ( i, (G.oriented_inc_matrix R o) i j)) x :
  by { congr' }
  ... = dot_product (vec_mul x (G.oriented_inc_matrix R o))
              ( (e : G.edge_set), dot_product ((G.oriented_inc_matrix R o) e) x) :
  by { rw dot_product_assoc }
  ... = dot_product (vec_mul x (G.oriented_inc_matrix R o))
              ((G.oriented_inc_matrix R o).mul_vec x) :
  by { congr' }                    -- ... = (x  N(o))  (N(o)  x)
  ... = dot_product (vec_mul x (G.oriented_inc_matrix R o))
              (vec_mul x (G.oriented_inc_matrix R o)) :
  by { rw mul_vec_transpose }          -- ... = (x  N(o))  (x  N(o))
  ... =  e : G.edge_set, (x (o.head e) - x (o.tail e)) ^ 2 :
  by { simp only [dot_product, vec_mul_oriented_inc_matrix],
     ring_nf }                  -- =  e, (x head(e) - x tail(e))  ^ 2

-- ## Signless Laplacian matrix Q
```

```
/-- `signless_laplace_matrix G` is the matrix `Q` of an `simple graph G` with ` i j  V` :
` | Q i j = G.degree i`, if `i = j`
` | Q i j = A i j`, otherwise. -/
def signless_laplace_matrix : matrix V V R
| i j := if i = j then G.degree i else G.adj_matrix R i j

@[simp]
lemma signless_laplace_matrix_apply {i j : V} :
 G.signless_laplace_matrix R i j = ite (i = j) (G.degree i) (G.adj_matrix R i j) := rfl

lemma signless_laplace_matrix_eq {i j : V} (H_eq : i = j) :
 G.signless_laplace_matrix R i j = G.degree i :=
by { rw [signless_laplace_matrix_apply, adj_matrix_apply],
    simp only [H_eq, if_true, eq_self_iff_true] }

lemma signless_laplace_matrix_neq {i j : V} (H_neq : i  j) :
 G.signless_laplace_matrix R i j = G.adj_matrix R i j :=
by simp only [signless_laplace_matrix_apply, adj_matrix_apply, H_neq, if_false]

-- Q = A + D
lemma signless_laplace_eq_degree_plus_adj :
 G.signless_laplace_matrix R = G.adj_matrix R + diagonal ( v, G.degree v) :=
begin
 ext,
 by_cases H : (i = j),
 { rw [G.signless_laplace_matrix_eq R H, dmatrix.add_apply, G.adj_matrix_eq R H,
     zero_add, H, diagonal_apply_eq] },
 { rw [G.signless_laplace_matrix_neq R H, dmatrix.add_apply, diagonal_apply_ne H, add_zero] }
end

-- 6. Q = M  M.
theorem signless_laplace_decomposition :
 G.signless_laplace_matrix R = G.inc_matrix R  (G.inc_matrix R) :=
begin
 ext,
 by_cases H_ij : i = j,
 { rw [signless_laplace_eq_degree_plus_adj, dmatrix.add_apply,
     G.adj_matrix_eq R H_ij, zero_add, mul_apply],
  simp only [H_ij, diagonal_apply_eq, degree_equals_sum_of_incidence_row,
         transpose_apply, inc_matrix_element_power_id] },
 { rw [signless_laplace_eq_degree_plus_adj, dmatrix.add_apply,
     diagonal_apply_ne H_ij, add_zero, mul_apply],
  by_cases H_adj : G.adj i j,
  { simp only [G.adj_matrix_adj R H_adj, transpose_apply, G.adj_sum_of_prod_inc_one R H_adj] },
  { simp only [G.adj_matrix_not_adj R H_adj, transpose_apply,
          G.inc_matrix_prod_non_adj R H_ij H_adj, sum_const_zero] } }
end

end simple_graph
```