lintrans

by D. Dyson

Centre Name:	The Duston School
Centre Number:	123456
Candidate Number:	123456

Contents

1 Analysis

One of the topics in the A Level Further Maths course is linear transformations, as represented by matrices. This is a topic all about how vectors move and get transformed in the plane. It's a topic that lends itself exceedingly well to visualization, but students often find it hard to visualize this themselves, and there is a considerable lack of good tools to provide visual intuition on the subject. There is the YouTube series *Essence of Linear Algebra* by 3blue1brown[7], which is excellent, but I couldn't find any good interactive visualizations.

My solution is to develop a desktop application that will allow the user to define 2×2 matrices and view these matrices and compositions thereof as linear transformations of a 2D plane. This will give students a way to get to grips with linear transformations in a more hands-on way, and will give teachers the ability to easily and visually show concepts like the determinant and invariant lines.

1.1 Computational Approach

This solution is particularly well suited to a computational approach since it is entirely focussed on visualizing transformations, which require complex mathematics to properly display. It will also have lots of settings to allow the user to configure aspects of the visualization. As previously mentioned, visualizing transformations in one's own head is difficult, so a piece of software to do it would be very valuable to teachers and learners, but current solutions are considerably lacking.

My solution will make use of abstraction by allowing the user to define a set of matrices which they can use in expressions. This allows them to use a matrix multiple times and they don't have to keep track of any of the numbers. All the actual processing and mathematics happens behind the scenes and the user never has to worry about it - they just compose their defined matrices into transformations. This abstraction allows the user to focus on exploring the transformations themselves without having to do any actual computations. This will make learning the subject much easier, as they will able to gain a visual intuition for linear transformations without worrying about computation until after they've built up that intuition.

I will also employ decomposition and modularization by breaking the project down into many smaller parts, such as one module to keep track of defined matrices, one module to validate and parse matrix expressions, one module for the main GUI, as well as sub-modules for the widgets and dialog boxes, etc. This decomposition allows for simpler project design, easier code maintenance (since module coupling is kept to a minimum, so bugs are isolated in their modules), inheritance of classes to reduce code repetition, and unit testing to inform development. I also intend this unit testing to be automated using GitHub Actions.

Selection will also be used widely in the application. The GUI will provide many settings for visualization, and these settings will need to be checked when rendering the transformation. For example, the user will have the option to render the determinant, so I will need to check this setting on every render cycle and only render the determinant parallelogram if the user has enabled that option. The app will have many options for visualization, which will be useful in learning, but if all these options were being rendered at the same time, then there would be too much information for the user to properly process, so I will let the user configure these display options to their liking and only render the things they want to be rendered.

Validation will also be prevalent because the matrix expressions will need to follow a strict format, which will be validated. The buttons to render and animate the matrix will only be clickable when the given expression is valid, so I will need to check this and update the buttons every time the text in the text box is changed. I will also need to parse matrix expressions so that I can evaluate them properly. All this validation ensures that crashes due to malformed input are practically impossible, and makes the user's life easier since they don't need to worry about if their input is in the right format - the app will tell them.

I will also make use of iteration, primarily in animation. I will have to re-calculate positions and

values to render everything for every frame of the animation and this will likely be done with a simple `for` loop. A `for` loop will allow me to just loop over every frame and use the counter variable as a way to measure how far through the animation we are on each frame. This is preferable to a `while` loop, since that would require me to keep track of which frame we're on with a separate variable.

Finally, the core of the application is visualization, so that will definitely be used a lot. I will have to calculate positions of points and lines based on given matrices, and when animating, I will also have to calculate these matrices based on the current frame. Then I will have to use the rendering capabilities of the GUI framework that I choose to render these calculated points and lines onto a widget, which will form the viewport of the main GUI. I may also have to convert between coordinate systems. I will have the origin in the middle with positive x going to the right and positive y going up, but I may need to convert that to standard computer graphics coordinates with the origin in the top left, positive x going to the right, and positive y going down. This visualization of linear transformations is the core component of the app and is the primary feature, so it is incredibly important.

1.2 Stakeholders

Stakeholders for my app include A Level Further Maths students and teachers, who learn and teach linear transformations respectively. They will be able to provide useful input as to what they would like to see in the app, and they can provide feedback on what they like and what I can add or improve. I already know from experience that linear transformations are tricky to visualize and a computer-based visualization would be useful. My stakeholders agreed with this. Many teachers said that a desktop app that could render and animate linear transformations would be useful in a classroom environment and students said that it would be helpful to have something that they could play around with at home and use to get to grips with matrices and linear transformations.

Some teachers also suggested that it would be useful to have an option to save and load sets of matrices. This would allow them to have a single save file containing some matrices, and then just load this file to use for demonstrations in the classroom. This would probably be quite easy to implement. I could just wrap all the relevant information into one object and use Python's `pickle` module to save the binary data to a file, and then load this data back into the app in a similar way.

My stakeholders agreed that being able to see incremental animation - where, for example, we apply matrix **A** to the current scene, pause, and then apply matrix **B** - would be beneficial. This would be a good demonstration of matrix multiplication being non-commutative. **AB** is not always equal to **BA**. Being able to see this in terms of animating linear transformations would be good for learning.

They also agreed that a tutorial on using the software would be useful, so I plan to implement this through an online written tutorial hosted with GitHub Pages, and perhaps a video tutorial as well. This would make the app much easier to use for people who have never seen it before. It wouldn't be a lesson on the maths itself, just a guide on how to use the software.

1.3 Research on existing solutions

There are actually quite a few web apps designed to help visualize 2D linear transformations but many of them are hard to use and lacking many features.

1.3.1 MIT 'Matrix Vector' Mathlet

Arguably the best app that I found was an MIT 'Mathlet' - a simple web app designed to help visualize a maths concept. This one is called 'Matrix Vector'[8] and allows the user to drag an input vector around the plane and see the corresponding output vector, transformed by a matrix that the user can define, although this definition is finicky since it involves sliders rather than keyboard input.

This app fails in two crucial ways in my opinion. It doesn't show the basis vectors or let the user drag them around, and the user can only define and therefore visualize a single matrix at once. This second problem was common among every solution I found, so I won't mention it again, but it is a big issue in my opinion and my app will allow for multiple matrices. I like the idea of having a draggable input vector and rendering its output, so I will probably have this feature in my app, but I also want the ability to define multiple matrices and be able to drag the basis vectors to visually define a matrix. Being able to drag the basis vectors will help build intuition, so I think this would greatly benefit the app.

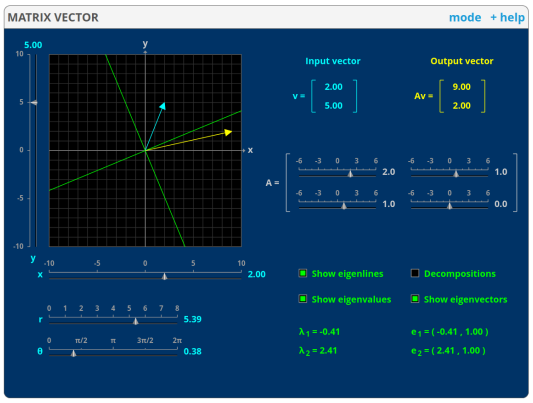


Figure 1.1: The MIT 'Matrix Vector' Mathlet

However, in the comments on this Mathlet, a user called 'David S. Bruce' suggested that the Mathlet should display the basis vectors, to which a user called 'hrm' (who I assume to be the 'H. Miller' to whom the copyright of the whole website is accredited) replied saying that this Mathlet is primarily focussed on eigenvectors, that it is perhaps badly named, and that displaying the basis vectors 'would make a good focus for a second Mathlet about 2×2 matrices'. This Mathlet does not exist. But I do like the idea of showing the eigenvectors and eigenlines, so I will definitely have that in my app. Showing the invariant lines or lack thereof will help with learning, since these are often hard to visualize.

1.3.2 Linear Transformation Visualizer

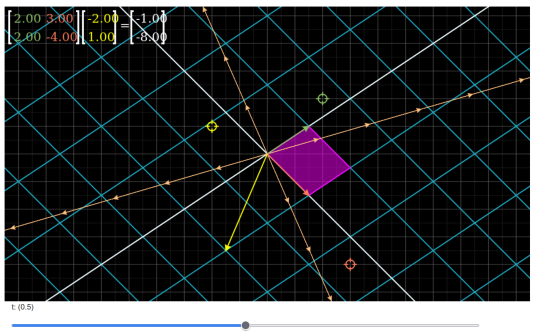


Figure 1.2: 'Linear Transformation Visualizer' halfway through an animation

Another web app that I found was one simply called 'Linear Transformation Visualizer' by Shad Sharma[22]. This one was similarly inspired by 3blue1brown's YouTube series. This app has the ability to render input and output vectors and eigenlines, but it can also render the determinant parallelogram; it allows the user to drag the basis vectors; and it has the option to snap vectors to the background grid, which is quite useful. It also implements a simple form of animation where the tips of the vectors move in straight lines from where they start to where they end, and the animation is controlled by dragging a slider labelled t. This isn't particularly intuitive.

I really like the vectors snapping to the grid, the input and output vectors, and rendering the determinant. This app also renders positive and negative determinants in different colours, which is really nice - I intend to use that idea in my own app, since it helps create understanding about negative determinants in terms of orientation changes. However, I think that the animation system here is flawed and not very easy to use. My animation will likely be a button, which just triggers an animation, rather than a slider. I also don't like the way vector dragging is handled. If you click anywhere on the grid, then the closest vector target (the final position of the target's associated vector) snaps to that location. I think it would be more intuitive to have to drag the vector from its current location to where you want it. This was also a problem with the MIT Mathlet.

1.3.3 Desmos app

One of the solutions I found was a Desmos app[6], which was quite hard to use and arguably overcomplicated. Desmos is not designed for this kind of thing - it's designed to graph pure mathematical functions - and it shows here. However, this app brings some really interesting ideas to the table, mainly functions. This app allows you to define custom functions and view them before and after the transformation. This is achieved by treating the functions parametrically as the set of points $(t, f(t))$ and then transforming each coordinate by the given matrix to get a new coordinate.

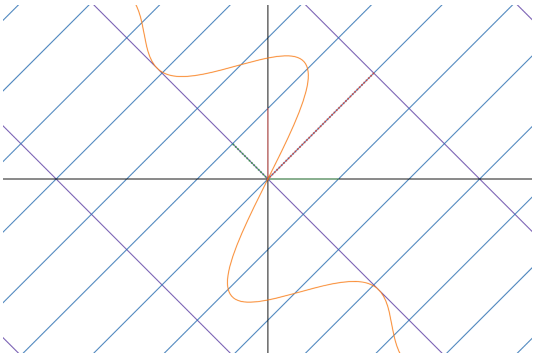


Figure 1.3: The Desmos app halfway through an animation, rendering $f(x) = \frac{\sin^2 x}{x}$ in orange

Desmos does this for every point and then renders the resulting transformed function parametrically. This is a really interesting technique and idea, but I'm not going to use it in my app. I don't think arbitrary functions fit with the linearity of the whole app, and I don't think it's necessary. It's just overcomplicating things, and rendering it on a widget would be tricky, because I'd have to render every point myself, possibly using something like OpenGL. It's just not worth implementing.

Additionally, this Desmos app makes things quite hard to see. It's hard to tell where any of the vectors are - they just get lost in the sea of grid lines. This image also hides some of the extra information. For instance, this image doesn't show the original function $f(x) = \frac{\sin^2 x}{x}$, only the transformed version. This app easily gets quite cluttered. I will give my vectors arrowheads to make them easily identifiable amongst the grid lines.

1.3.4 Visualizing Linear Transformations

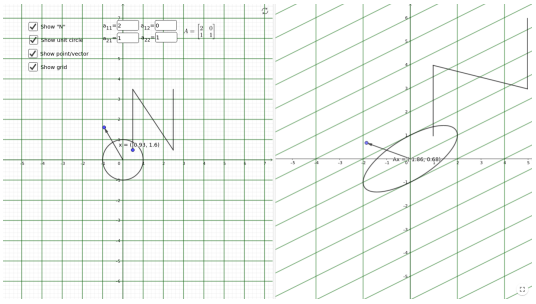


Figure 1.4: The GeoGebra applet rendering its default matrix

The last solution that I want to talk about is a GeoGebra applet simply titled 'Visualizing Linear Transformations'[10]. This applet has input and output vectors, original and transformed grid lines, a unit circle, and the letter N. It allows the user to define a matrix as 4 numbers and view the aforementioned N (which the user can translate to anywhere on the grid), the unit circle, the input/output vectors, and the grid lines. It also has the input vector snapping to integer coordinates, but that's a standard part of GeoGebra.

I've already talked about most of these features but the thing I wanted to talk about here is the N. I don't particularly want the letter N to be a prominent part of my own app, but I really like the idea of being able to define a custom polygon and see how that polygon gets transformed by a given transformation. I think that would really help with building intuition and it shouldn't be too hard to implement.

1.4 Essential features

The primary aim of this application is to visualize linear transformations, so this will obviously be the centre of the app and an essential feature. I will have a widget which can render a background grid and a second version of the grid, transformed according to a user-defined matrix expression. This is necessary because it is the entire purpose of the app. It's designed to visualize linear transformations and would be completely useless without this visual component. I will give the user the ability to render a custom matrix expression containing matrices they have previously defined, as well as reset the canvas to the default identity matrix transformation. This will obviously require an input box to enter the expression, a render button, a reset button, and various dialog boxes to define matrices in different ways. I want the user to be able to define a matrix as a set of 4 numbers, and by dragging the basis vectors i and j. These dialogs will allow the user to define new matrices to be used in expressions, and having multiple ways to do it will make it easier, and will aid learning.

Another essential feature is animation. I want the user to be able to smoothly animate between matrices. I see two options for how this could work. If **C** is the matrix for the currently displayed transformation, and **T** is the matrix for the target transformation, then we could either animate from **C** to **T** or we could animate from **C** to **TC**. I would probably call these transitional and applicative animation respectively. Perhaps I'll give the user the option to choose which animation method they want to use. I might even have an option for sequential animation, where the user can define a sequence of matrices, perhaps separated with commas or semicolons, and the app will animate through the sequence, applying one at a time. Sequential animation would be nice, but is not crucial.

Either way, animation is used in most of the alternative solutions that I found, and it's a great way to build intuition, by allowing students to watch the transformation happen in real time. Compared to simply rendering the transformations, animating them would profoundly benefit learning, and since that's the main aim of the project, I think animation is a necessary part of the app.

Something that I thought was a big problem in every alternative solution I found was the fact that the user could only visualize a single matrix at once. I see this as a fatal flaw and I will allow the user to define 25 different matrices (all capital letters except **I** for the identity matrix) and use all of them in expressions. This will allow teachers to define multiple matrices and then just change the expression to demonstrate different concepts rather than redefine a new transformation every time. It will also make things easier for students as it will allow them to visualize compositions of different matrix transformations without having to do any computations themselves.

Additionally, being able to show information on the currently displayed matrix is an essential tool for learning. Rendering things like the determinant parallelogram and the invariant lines of the transformation will greatly assist with learning and building understanding, so I think that having the option to render these attributes of the currently displayed transformation is necessary for success.

1.5 Limitations

The main limitation in this app is likely to be drawing grid lines. Most transformations will be fine but in some cases, the app will be required to draw potentially thousands of grid lines on the canvas and this will probably cause noticeable lag, especially in the animations. I will have to artificially limit the number of grid lines that can be drawn on the screen. This won't look fantastic, because it means that the grid lines will only extend a certain distance from the origin, but it's an inherent limitation of computers. Perhaps if I was using a faster, compiled language like C++ rather than Python, this processing would happen faster and I could render more grid lines, but it's impossible to render all the grid lines and any implementation of this idea must limit them for performance.

An interesting limitation is that I don't think I'll implement panning. I suspect that I'll have to convert between coordinate systems and having the origin in the centre of the canvas will probably make the code much simpler. Also, linear transformations always leave the origin fixed, so always having it in the centre of the canvas seems thematically appropriate. Panning is certainly an option - the Desmos solution in §1.3.3 and GeoGebra solution in §1.3.4 both allow panning as a default part

of Desmos and GeoGebra respectively, for example - but I don't think I'll implement it myself. I just don't think it's worth it.

I'm also not going to do any work with 3D linear transformations. 3D transformations are often harder to visualize and thus it would make sense to target them in an app like this, designed to help with learning and intuition, but 3D transformations are also harder to code. I would have to use a full graphics package rather than a simple widget, and I think it would be too much work for this project and I wouldn't be able to do it in the time frame. It's definitely a good idea, but I'm currently incapable of creating an app like that.

There are other limitations inherent to matrices. For instance, it's impossible to take an inverse of a singular matrix. There's nothing I can do about that without rewriting most of mathematics. Matrices can also only represent linear transformations. There's definitely a market for an app that could render any arbitrary transformation from $\mathbb{R}^2 \to \mathbb{R}^2$ - I know I'd want an app like that - but matrices can only represent linear transformations, so those are the only kind of transformations that I'll be looking at with this project.

1.6 Hardware and software requirements

1.6.1 Hardware

Hardware requirements for the project are the same between the release and development environments and they're quite simple. I expect the app to require a processor with at least 1 GHz clock speed, `$BINARY_SIZE` free disk space, and about 1 GB of available RAM. The processor and RAM requirements are needed by the Python runtime and mainly by Qt5 - the GUI library I'll be using. The `$BINARY_SIZE` disk space is just for the executable binary that I'll compile for the public release. The code itself is less than 1 MB, but the compiled binary has to package all the dependencies and the entire CPython runtime to allow it to run on systems that don't have that, so the file size is much bigger.

I will also require that the user has a monitor that is at least 1920×1080 pixels in resolution. This isn't necessarily required, because the app will likely run in a smaller window, but a HD monitor is highly recommended. This allows the user to go fullscreen if they want to, and it gives them enough resolution to easily see everything in the app. A large, wall-mounted screen is also highly recommended for use in the classroom, although this is common among schools.

I will also require a keyboard with all standard Latin alphabet characters. This is because the matrices are defined as uppercase Latin letters. Any UK or US keyboard will suffice for this. The app will also require a mouse with at least one button. I don't intend to have right click do anything, so only the primary mouse button is required, although getting a single button mouse to actually work on modern computers is probably quite a challenge. A separate mouse is not strictly required - a laptop trackpad is equally sufficient.

1.6.2 Software

Software requirements differ slightly between release and development, although everything that the release environment requires is also required by the development environment. I will require a modern operating system - namely Windows 10 or later, macOS 10.9 'Mavericks'[1] or later, or any modern Linux distro[2]. Basically, it just requires an operating system that is compatible with Python 3.8 or higher as well as Qt5, since I'll be using these in the project. Of course, Qt5 will need to be installed on the user's computer, although it's standard pretty much everywhere these days.

[1] Python 3.8 or higher won't compile on any earlier versions of macOS[16]

[2] Specifying a Linux version is practically impossible. Python 3.8 or higher is available in many package repositories, but all modern Python versions will compile on any modern distro. Qt5 is available in many package repositories and can be compiled on any `x86` or `x86_64` generic Linux machine with `gcc` version 5 or later[17]

Python won't actually be required for the end user, because I will be compiling the app into a standalone binary executable for release, and this binary will contain the required Python runtime and dependencies. However, if the user wishes to download and run the source code themself, then they will need Python 3.8 or higher and the package dependencies: `numpy`, `nptyping`, and `pyqt5`. These can be automatically installed with the command `python -m pip install -r requirements.txt` from the root of the repository, although the whole project will be an installable Python package, so using `pip install -e .` will be preferred.

`numpy` is a maths library that allows for fast matrix maths; `nptyping` is used by `mypy` for type-checking and isn't actually a runtime dependency but the imports in the `typing` module fail if it's not installed at runtime[3]; and `pyqt5` is a library that just allows interop between Python and Qt5, which is originally a C++ library.

In the development environment, I use PyCharm for actually writing my code, and I use a virtual environment to isolate my project dependencies. There are also some development dependencies listed in the file `dev_requirements.txt`. They are: `mypy`, `pyqt5-stubs`, `flake8`, `pycodestyle`, `pydocstyle`, and `pytest`. `mypy` is a static type checker[4]; `pyqt5-stubs` is a collection of type annotations for the PyQt5 API for `mypy` to use; `flake8`, `pycodestyle`, and `pydocstyle` are all linters; and `pytest` is a unit testing framework. I use these libraries to make sure my code is good quality and actually working properly during development.

1.7 Success criteria

The main aim of the app is to help teach students about linear transformations. As such, the primary measure of success will be letting teachers get to grips with the app and then asking if they would use it in the classroom or recommend it to students to use at home.

Additionally, the app must fulfil some basic requirements:

1. It must allow the user to define multiple matrices in at least two different ways (numerically and visually)
2. It must be able to validate arbitrary matrix expressions
3. It must be able to render any valid matrix expression
4. It must be able to animate any valid matrix expression
5. It must be able to apply a matrix expression to the current scene and animate this (animate from **C** to **TC**, and perhaps do sequential animation)
6. It must be able to display information about the currently rendered transformation (determinant, eigenlines, etc.)
7. It must be able to save and load sessions (defined matrices, display settings, etc.)
8. It must allow the user to define and transform arbitrary polygons

Defining multiple matrices is a feature that I thought was lacking from every other solution I researched, and I think it would make the app much easier to use, so I think it's necessary for success. Validating matrix expressions is necessary because if the user tries to render an expression that doesn't make sense, has an undefined matrix, or contains the inverse of a singular matrix, then we have to disallow that or else the app will crash.

Visualizing matrix expressions as linear transformations is the core part of the app, so basic rendering of them is definitely a requirement for success. Animating these expressions is also a pretty crucial part of the app, so I would consider this necessary for success. Displaying the information of a matrix

[3] These `nptyping` imports are needed for type annotations all over the code base, so factoring them out is not feasible

[4] Python has weak, dynamic typing with optional type annotations but `mypy` enforces these static type annotations

transformation is also very useful for building understanding, so I would consider this needed to succeed.

Saving and loading isn't strictly necessary for success, but it is a standard part of many apps, so will likely be expected by users, and it will benefit the app by allowing teachers to plan lessons in advance and save the matrices they've defined for that lesson to be loaded later.

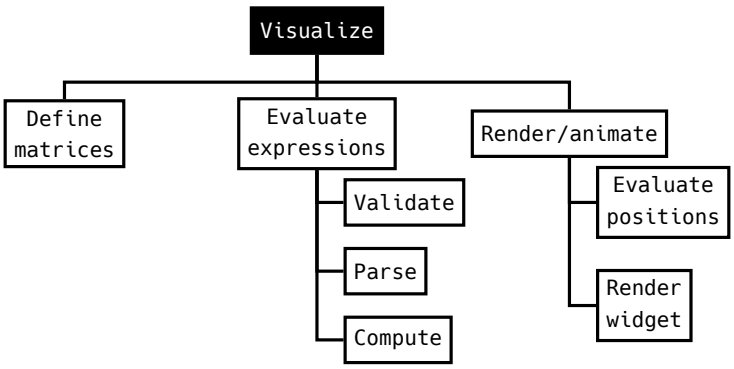
Transforming polygons is the lowest priority item on this list and will likely be implemented last, but it would definitely benefit learning. I wouldn't consider it necessary for success, but it would be very good to include, and it's certainly a feature that I want to have.

If the majority of teachers would use and/or recommend the app and it meets all of these points, then I will consider the app as a whole to be a success.

2 Design

2.1 Problem decomposition

I have decomposed the problem of visualization as follows:



Defining matrices is key to visualization because we need to have matrices to actually visualize. This is a key part of the app, and the user will be able to define multiple separate matrices numerically and visually using the GUI.

Evaluating expressions is another key part of the app and can be further broken down into validating, parsing, and computing the value. Validating an expression simply consists of checking that it adheres to a set of syntax rules for matrix expressions, and that it only contains matrices which have already been defined. Parsing consists of breaking an expression down into tokens, which are then much easier to evaluate. Computing the expression with these tokens is then just a series of simple operations, which will produce a final matrix at the end.

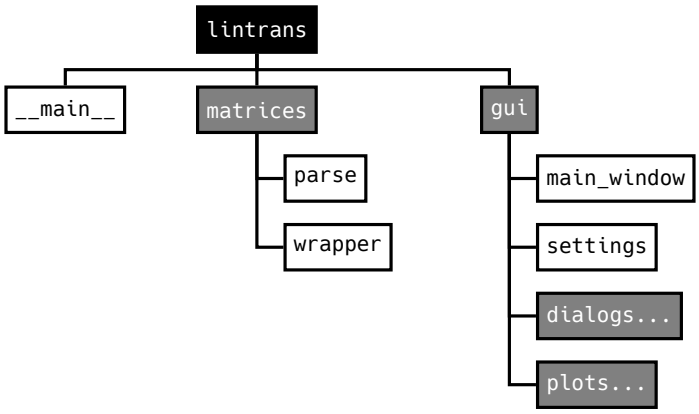
Rendering and animating will likely be the largest part in reality, but I've only decomposed it into simple blocks here. Evaluating positions involves evaluating the matrix expression that the user has input and using the columns of the resultant matrix to find the new positions of the basis vectors, and then extrapolating this for the rest of the plane. Rendering onto the widget is likely to be quite complicated and framework-dependent, so I've abstracted away the details for brevity here. Rendering will involve using the previously calculated values to render grid lines and vectors. Animating will probably be a `for` loop which just renders slightly different matrices onto the widget and sleeps momentarily between frames.

I have deliberately broken this problem down into parts that can be easily translated into modules in my eventual coded solution. This is simply to ease the design and development process, since now I already know my basic project structure. This problem could've been broken down into the parts that the user will directly interact with, but that would be less useful to me when actually starting development, since I would then have to decompose the problem differently to write the actual code.

2.2 Structure of the solution

2.2.1 The main project

I have decomposed my solution like so:



The `lintrans` node is simply the root of the whole project. `__main__` is the Python way to make the project executable as `python -m lintrans` on the command line. For release, I will package it into a standalone binary executable.

`matrices` is the package that will allow the user to define, validate, parse, evaluate, and use matrices. The `parse` module will contain functions to validate matrix expressions - likely using regular expressions - and functions to parse matrix expressions. It will not know which matrices are defined, so validation will be naïve and evaluation will be elsewhere. The `wrapper` module will contain a `MatrixWrapper` class, which will hold a dictionary of matrix names and values. It is this class which will have aware validation - making sure that all matrices are actually defined - as well the ability to evaluate matrix expressions, in addition to its basic behaviour of setting and getting matrices. This `matrices` package will also have a `create_rotation_matrix` function that will generate a rotation matrix from an angle using the formula $\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$. It will be in the `wrapper` module since it's related to defining and manipulating matrices, but it will be exported and accessible as `lintrans.matrices.create_rotation_matrix`.

`gui` is the package that will contain all the frontend code for everything GUI-related. `main_window` is the module that will contain a `LintransMainWindow` class, which will act as the main window of the application and have an instance of `MatrixWrapper` to keep track of which matrices are defined and allow for evaluation of matrix expressions. It will also have methods for rendering and animating matrix expressions, which will be connected to buttons in the GUI. This module will also contain a simple `main()` function to instantiate and launch the application GUI.

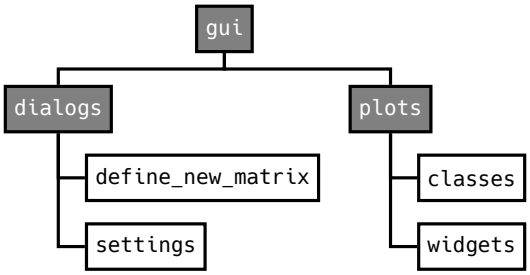
The `settings` module will contain a `DisplaySettings` dataclass[5] that will represent the settings for visualizing transformations. The `LintransMainWindow` class will have an instance of this class and check against it when rendering things. The user will be able to open a dialog to change these display settings, which will update the main window's instance of this class.

The `settings` module will also have a `GlobalSettings` class, which will represent the global settings for the application, such as the logging level, where to save the logs, whether to ask the user if they want to be prompted with a tutorial whenever they open the app, etc. This class will have defaults for everything, but the constructor will try to read these settings from a config file if possible. This allows for persistent settings between sessions. This config file will be `~/.config/lintrans.conf` on Unix-like systems, including macOS, and `C:\Users\%USER%\AppData\Roaming\lintrans\config.txt` on Windows. This difference is to remain consistent with operating system conventions[6].

[5] This is the Python equivalent of a `struct` or `record` in other languages

[6] And also to avoid confusing Windows users with a `.conf` file

2.2.2 The `gui` subpackages



The `dialogs` subpackage will contain modules with different dialog classes. It will have a `define_new_matrices` module, which will have a `DefineDialog` abstract superclass. It will also contain classes that inherit from this superclass and provide dialogs for defining new matrices visually, numerically, and as an expression in terms of other matrices. Additionally, this subpackage will contain a `settings` module, which will provide a `SettingsDialog` superclass and a `DisplaySettingsDialog` class, which will allow the user to configure the aforementioned display settings. It will also have a `GlobalSettingsDialog` class, which will similarly allow the user to configure the app's global settings through a dialog.

The `plots` subpackage will have a `classes` module and a `widgets` module. The `classes` module will have the abstract superclasses `BackgroundPlot` and `VectorGridPlot`. The former will provide helped methods to convert between coordinate systems and draw the background grid, while the latter will provide helper methods to draw transformations and their components. It will have `point_i` and `point_j` attributes and will provide methods to draw the transformed version of the grid, the vectors and their arrowheads, the eigenlines of the transformation, etc. These methods can then be called from the Qt5 `paintEvent` handler which will be declared abstract and must therefore be implemented by all subclasses.

The `plots` subpackage will also contain a `widgets` module, which will have the classes `VisualizeTransformationWidget` and `DefineVisuallyWidget`, both of which will inherit from `VectorGridPlot`. They will both implement their own `paintEvent` handler to actually draw the respective widgets, and `DefineVisuallyWidget` will also implement handlers for mouse events, allowing the user to drag around the basis vectors.

It's also worth noting here that I don't currently know how I'm going to implement the transformation of arbitrary polygons. It will likely consist of an attribute in `VisualizeTransformationWidget` which is a list of points, and these points can be dragged around with mouse event handlers and then the transformed versions can be rendered, but I'm not yet sure about how I'm going to implement it.

2.3 Algorithm design

This section will be completed later.

2.4 Usability features

My main concern in terms of usability is colour. In the 3blue1brown videos on linear algebra, red and green are used for the basis vectors, but these colours are often hard to distinguish in most common forms of colour blindness. The most common form is deuteranopia[25], which makes red and green look incredibly similar. I will use blue and red for my basis vectors. These colours are easy to distinguish for people with deuteranopia and protanopia - the two most common forms of colour blindness. Tritanopia makes it harder to distinguish blue and yellow, but my colour scheme is still be accessible for people with tritanopia, as red and blue are very distinct in this form of colour blindness.

I will probably use green for the eigenvectors and eigenlines, which will be hard to distinguish from the red basis vector for people with red-green colour blindness, but I think that the basis vectors and

eigenvectors/eigenlines will look physically different enough from each other that the colour shouldn't be too much of a problem. Additionally, I will use a tool called Color Oracle[11] to make sure that my app is accessible to people with different forms of colour blindness[7].

Another solution would be to have one default colour scheme, and allow the user to change the colour scheme to something more accessible for colour blind people, but I don't see the point in this. I think it's easier for colour blind people to just have the main colour scheme be accessible, and it's not really an inconvenience to non-colour blind people, so I think this is the best option.

The layout of my app will be self-consistent and follow standard conventions. I will have a menu bar at the top of the main window for actions like saving and loading, as well as accessing the tutorial (which will also be accessible by pressing `F1` at any point) and documentation. The dialogs will always have the confirm button in the bottom right and the cancel button just to the left of that. They will also have the matrix name drop-down on the left. This consistency will make the app easier to learn and understand.

I will also have hotkeys for everything that can have hotkeys - buttons, checkboxes, etc. This makes my life easier, since I'm used to having hotkeys for everything, and thus makes the app faster to test because I don't need to click everything. This also makes things easier for other people like me, who prefer to stay at the keyboard and not use the mouse. Obviously a mouse will be required for things like dragging basis vectors and polygon vertices, but hotkeys will be available wherever possible to help people who don't like using the mouse or find it difficult.

2.5 Variables and validation

This project won't actually have many variables. The main ones will be instance attributes on the `LintransMainWindow` class. It will have a `MatrixWrapper` instance, a `DisplaySettings` instance, and a `GlobalSettings` instance. These will handle the matrices and various settings respectively. Having these as instance attributes allows them to be referenced from any method in the class, and Qt5 uses lots of slots (basically callback methods) and handlers, so it's good to be able to access the attributes I need right there rather than having to pass them around from method to method.

The `MatrixWrapper` class will have a dictionary of names and matrices. The names will be single letters[8] and the matrices will be of type `MatrixType`. This will be a custom type alias representing a 2×2 `numpy` array of floats. When setting the values for these matrices, I will have to manually check the types. This is because Python has weak typing, and if we got, say, an integer in place of a matrix, then operations would fail when trying to evaluate a matrix expression, and the program would crash. To prevent this, we have to validate the type of every matrix when it's set. I have chosen to use a dictionary here because it makes accessing a matrix by its name easier. We don't have to check against a list of letters and another list of matrices, we just index into the dictionary.

The settings dataclasses will have instance attributes for each setting. Most of these will be booleans, since they will be simple binary options like *Show determinant*, which will be represented with checkboxes in the GUI. The `DisplaySettings` dataclass will also have an attribute of type `int` representing the time in milliseconds to pause during animations.

The `DefineDialog` superclass have a `MatrixWrapper` instance attribute, which will be a parameter in the constructor. When `LintransMainWindow` spawns a definition dialog (which subclasses `DefineDialog`), it will pass in a copy of its own `MatrixWrapper` and connect the `accepted` signal for the dialog. The slot (method) that this signal is connected to will get called when the dialog is closed with the *Confirm* button[9]. This allows the dialog to mutate its own `MatrixWrapper` object and then the main window can copy that mutated version back into its own instance attribute when the user confirms the change. This reduces coupling and makes everything easier to reason about and debug, as well as reducing

[7] I actually had to clone a fork of this project[1] to get it working on Ubuntu 20.04 and adapt it slightly to create a working `jar` file

[8] I would make these `char` but Python only has a `str` type for strings

[9] Actually when the dialog calls `.accept()`. The `Confirm` button is actually connected to a method which first takes the info and updates the instance `MatrixWrapper`, and then calls `.accept()`

the number of bugs, since the classes will be independent of each other. In another language, I could pass a pointer to the wrapper and let the dialog mutate it directly, but this is potentially dangerous, and Python doesn't have pointers anyway.

Validation will also play a very big role in the application. The user will be able to enter matrix expressions and these must be validated. I will define a BNF schema and either write my own RegEx or use that BNF to programmatically generate a RegEx. Every matrix expression input will be checked against it. This is to ensure that the matrix wrapper can actually evaluate the expression. If we didn't validate the expression, then the parsing would fail and the program could crash. I've chosen to use a RegEx here rather than any other option because it's the simplest. Creating a RegEx can be difficult, especially for complicated patterns, but it's then easier to use it. Also, Python can compile a RegEx pattern, which makes it much faster to match against, so I will compile the pattern at initialization time and just compare expressions against that pre-compiled pattern, since we know it won't change at runtime.

Additionally, the buttons to render and animate the current matrix expression will only be enabled when the expression is valid. Textboxes in Qt5 emit a `textChanged` signal, which can be connected to a slot. This is just a method that gets called whenever the text in the textbox is changed, so I can use this method to validate the input and update the buttons accordingly. An empty string will count as invalid, so the buttons will be disabled when the box is empty.

I will also apply this matrix expression validation to the textbox in the dialog which allows the user to define a matrix as an expression involving other matrices, and I will validate the input in the numeric definition dialog to make sure that all the inputs are floats. Again, this is to prevent crashes, since a matrix with non-number values in it will likely crash the program.

2.6 Iterative test data

In unit testing, I will test the validation, parsing, and generation of rotation matrices from an angle. I will also unit test the utility functions for the GUI, like `is_valid_float`.

For the validation of matrix expressions, I will have data like the following:

Valid	Invalid
`"A"`	`""`
`"AB"`	`"A^"`
`"-3.4A"`	`"rot()"`
`"A^2"`	`"A^{2"`
`"A^T"`	`"^12"`
`"A^{-1}"`	`"A^3.2"`
`"rot(45)"`	`"A^B"`
`"3A^{12}"`	`".A"`
`"2B^2+A^TC^{-1}"`	`"--A"`
`"3.5A^45.6rot(19.2^T-B^-14.1C^5"`	`"A--B"`

This list is not exhaustive, mostly to save space and time, but the full unit testing code is included in appendix B.

The invalid expressions presented here have been chosen to be almost valid, but not quite. They are edge cases. I will also test blatantly invalid expressions like `"This is a matrix expression"` to make sure the validation works.

Here's an example of some test data for parsing:

Input	Expected
`"A"`	`[[("", "A", "")]]`
`"AB"`	`[[("", "A", ""), ("", "B", "")]]`
`"2A+B^2"`	`[[("2", "A", ""), ("", "B", "2")]]`
`"3A^T2.4B^{-1}-C"`	`[[("3", "A", "T"), ("2.4", "B", "-1")], [("-1", "C", "")]]`

The parsing output is pretty verbose and this table doesn't have enough space for most of the more complicated inputs, so here's a monster one:

```
"2.14A^{3} 4.5rot(14.5)^-1 + 8.5B^T 5.97C^14 - 3.14D^{-1} 6.7E^T"
```

which should parse to give:

```
[[("2.14", "A", "3"), ("4.5", "rot(14.5)", "-1")], [("8.5", "B", "T"), ("5.97", "C", "14")],
                        [("-3.14", "D", "-1"), ("6.7", "E", "T")]]
```

Any invalid expression will also raise a parse error, so I will check every invalid input previously mentioned and make sure it raises the appropriate error.

Again, this section is brief to save space and time. All unit tests are included in appendix B.

2.7 Post-development test data

This section will be completed later.

2.8 Issues with testing

Since `lintrans` is a graphical application about visualizing things, it will be mainly GUI focussed. Unfortunately, unit testing GUIs is a lot harder than unit testing library or API code. I don't think there's any way to easily and reliably unit test a graphical interface, so my unit tests will only cover the backend code for handling matrices. Testing the GUI will be entirely manual; mostly defining matrices, thinking about what I expect them to look like, and then making sure they look like that. I don't see a way around this limitation. I will make my backend unit tests very thorough, but testing the GUI can only be done manually.

3 Development

Please note, throughout this section, every code snippet will have two comments at the top. The first is the git commit hash that the snippet was taken from[10]. The second comment is the file name. The line numbers of the snippet reflect the line numbers of the file from where the snippet was taken. After a certain point, I introduced copyright comments at the top of every file. These are always omitted here.

3.1 Matrices backend

3.1.1 `MatrixWrapper` class

The first real part of development was creating the `MatrixWrapper` class. It needs a simple instance dictionary to be created in the constructor, and it needs a way of accessing the matrices. I decided to use Python's `__getitem__()` and `__setitem__()` special methods[15] to allow indexing into a `MatrixWrapper` object like `wrapper['M']`. This simplifies using the class.

```
# 29ec1fedbf307e3b7ca731c4a381535fec899b0b
# src/lintrans/matrices/wrapper.py
```

```python
"""A module containing a simple MatrixWrapper class to wrap matrices and context."""

import numpy as np

from lintrans.typing import MatrixType


class MatrixWrapper:
    """A simple wrapper class to hold all possible matrices and allow access to them."""

    def __init__(self):
        """Initialise a MatrixWrapper object with a matrices dict."""
        self._matrices: dict[str, MatrixType | None] = {
            'A': None, 'B': None, 'C': None, 'D': None,
            'E': None, 'F': None, 'G': None, 'H': None,
            'I': np.eye(2),  # I is always defined as the identity matrix
            'J': None, 'K': None, 'L': None, 'M': None,
            'N': None, 'O': None, 'P': None, 'Q': None,
            'R': None, 'S': None, 'T': None, 'U': None,
            'V': None, 'W': None, 'X': None, 'Y': None,
            'Z': None
        }

    def __getitem__(self, name: str) -> MatrixType | None:
        """Get the matrix with `name` from the dictionary.

        Raises:
            KeyError:
                If there is no matrix with the given name
        """
        return self._matrices[name]

    def __setitem__(self, name: str, new_matrix: MatrixType) -> None:
        """Set the value of matrix `name` with the new_matrix.

        Raises:
            ValueError:
                If `name` isn't a valid matrix name
        """
        name = name.upper()

        if name == 'I' or name not in self._matrices:
            raise NameError('Matrix name must be a capital letter and cannot be "I"')
```

[10] A history of all commits can be found in the GitHub repository[2]

```
44
45              self._matrices[name] = new_matrix
```

This code is very simple. The constructor (`__init__()`) creates a dictionary of matrices which all start out as having no value, except the identity matrix **I**. The `__getitem__()` and `__setitem__()` methods allow the user to easily get and set matrices just like a dictionary, and `__setitem__()` will raise an error if the name is invalid. This is a very early prototype, so it doesn't validate the type of whatever the user is trying to assign it to yet. This validation will come later.

I could make this class subclass `dict`, since it's basically just a dictionary at this point, but I want to extend it with much more functionality later, so I chose to handle the dictionary stuff myself.

I then had to write unit tests for this class, and I chose to do all my unit tests using a framework called `pytest`.

```
# 29ec1fedbf307e3b7ca731c4a381535fec899b0b
# tests/test_matrix_wrapper.py

"""Test the MatrixWrapper class."""

import numpy as np
import pytest
from lintrans.matrices import MatrixWrapper

valid_matrix_names = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
test_matrix = np.array([[1, 2], [4, 3]])


@pytest.fixture
def wrapper() -> MatrixWrapper:
    """Return a new MatrixWrapper object."""
    return MatrixWrapper()


def test_get_matrix(wrapper) -> None:
    """Test MatrixWrapper.__getitem__()."""
    for name in valid_matrix_names:
        assert wrapper[name] is None

    assert (wrapper['I'] == np.array([[1, 0], [0, 1]])).all()


def test_get_name_error(wrapper) -> None:
    """Test that MatrixWrapper.__getitem__() raises a KeyError if called with an invalid name."""
    with pytest.raises(KeyError):
        _ = wrapper['bad name']
        _ = wrapper['123456']
        _ = wrapper['Th15 Is an 1nV@l1D n@m3']
        _ = wrapper['abc']


def test_set_matrix(wrapper) -> None:
    """Test MatrixWrapper.__setitem__()."""
    for name in valid_matrix_names:
        wrapper[name] = test_matrix
        assert (wrapper[name] == test_matrix).all()


def test_set_identity_error(wrapper) -> None:
    """Test that MatrixWrapper.__setitem__() raises a NameError when trying to assign to I."""
    with pytest.raises(NameError):
        wrapper['I'] = test_matrix


def test_set_name_error(wrapper) -> None:
    """Test that MatrixWrapper.__setitem__() raises a NameError when trying to assign to an invalid name."""
    with pytest.raises(NameError):
        wrapper['bad name'] = test_matrix
        wrapper['123456'] = test_matrix
```

```
52          wrapper['Th15 Is an 1nV@l1D n@m3'] = test_matrix
53          wrapper['abc'] = test_matrix
```

These tests are quite simple and just ensure that the expected behaviour works the way it should, and that the correct errors are raised when they should be. It verifies that matrices can be assigned, that every valid name works, and that the identity matrix **I** cannot be assigned to.

The function decorated with `@pytest.fixture` allows functions to use a parameter called `wrapper` and `pytest` will automatically call this function and pass it as that parameter. It just saves on code repetition.

3.1.2 Rudimentary parsing and evaluating

This first thing I did here was improve the `__setitem__()` and `__getitem__()` methods to validate input and easily get transposes and simple rotation matrices.

```
# f89fc9fd8d5917d07557fc50df3331123b55ad6b
# src/lintrans/matrices/wrapper.py
```

```
60      def __setitem__(self, name: str, new_matrix: MatrixType) -> None:
61          """Set the value of matrix `name` with the new_matrix.
62
63          :param str name: The name of the matrix to set the value of
64          :param MatrixType new_matrix: The value of the new matrix
65          :rtype: None
66
67          :raises NameError: If the name isn't a valid matrix name or is 'I'
68          """
69          if name not in self._matrices.keys():
70              raise NameError('Matrix name must be a single capital letter')
71
72          if name == 'I':
73              raise NameError('Matrix name cannot be "I"')
74
75          # All matrices must have float entries
76          a = float(new_matrix[0][0])
77          b = float(new_matrix[0][1])
78          c = float(new_matrix[1][0])
79          d = float(new_matrix[1][1])
80
81          self._matrices[name] = np.array([[a, b], [c, d]])
```

In this method, I'm now casting all the values to floats. This is very simple validation, since this cast will raise `ValueError` if it fails to cast the value to a float. I should've declared `:raises ValueError:` in the docstring, but this was an oversight at the time.

```
# f89fc9fd8d5917d07557fc50df3331123b55ad6b
# src/lintrans/matrices/wrapper.py
```

```
27      def __getitem__(self, name: str) -> Optional[MatrixType]:
28          """Get the matrix with the given name.
29
30          If it is a simple name, it will just be fetched from the dictionary.
31          If the name is followed with a 't', then we will return the transpose of the named matrix.
32          If the name is 'rot()', with a given angle in degrees, then we return a new rotation matrix with that angle.
33
34          :param str name: The name of the matrix to get
35          :returns: The value of the matrix (may be none)
36          :rtype: Optional[MatrixType]
37
38          :raises NameError: If there is no matrix with the given name
39          """
40          # Return a new rotation matrix
```

```
41          match = re.match(r'rot\((\d+)\)', name)
42          if match is not None:
43              return create_rotation_matrix(float(match.group(1)))
44
45          # Return the transpose of this matrix
46          match = re.match(r'([A-Z])t', name)
47          if match is not None:
48              matrix = self[match.group(1)]
49
50              if matrix is not None:
51                  return matrix.T
52              else:
53                  return None
54
55          if name not in self._matrices:
56              raise NameError(f'Unrecognised matrix name "{name}"')
57
58          return self._matrices[name]
```

This `__getitem__()` method now allows for easily accessing transposes and rotation matrices by checking input with regular expressions. This makes getting matrices easier and thus makes evaluating full expressions simpler.

The `create_rotation_matrix()` method is also defined in this file and just uses the $\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$ formula from before:

```
# f89fc9fd8d5917d07557fc50df3331123b55ad6b
# src/lintrans/matrices/wrapper.py

158 def create_rotation_matrix(angle: float) -> MatrixType:
159     """Create a matrix representing a rotation by the given number of degrees anticlockwise.
160
161     :param float angle: The number of degrees to rotate by
162     :returns MatrixType: The resultant rotation matrix
163     """
164     rad = np.deg2rad(angle)
165     return np.array([
166         [np.cos(rad), -1 * np.sin(rad)],
167         [np.sin(rad), np.cos(rad)]
168     ])
```

At this stage, I also implemented a simple parser and evaluator using regular expressions. It's not great and it's not very flexible, but it can evaluate simple expressions.

```
# f89fc9fd8d5917d07557fc50df3331123b55ad6b
# src/lintrans/matrices/wrapper.py

83      def parse_expression(self, expression: str) -> MatrixType:
84          """Parse a given expression and return the matrix for that expression.
85
86          Expressions are written with standard LaTeX notation for exponents. All whitespace is ignored.
87
88          Here is documentation on syntax:
89              A single matrix is written as 'A'.
90              Matrix A multiplied by matrix B is written as 'AB'
91              Matrix A plus matrix B is written as 'A+B'
92              Matrix A minus matrix B is written as 'A-B'
93              Matrix A squared is written as 'A^2'
94              Matrix A to the power of 10 is written as 'A^10' or 'A^{10}'
95              The inverse of matrix A is written as 'A^-1' or 'A^{-1}'
96              The transpose of matrix A is written as 'A^T' or 'At'
97
98          :param str expression: The expression to be parsed
99          :returns MatrixType: The matrix result of the expression
100
101         :raises ValueError: If the expression is invalid, such as an empty string
102         """
```

```
103             if expression == '':
104                 raise ValueError('The expression cannot be an empty string')
105 
106             match = re.search(r'[^-+A-Z^{}rot()\d.]', expression)
107             if match is not None:
108                 raise ValueError(f'Invalid character "{match.group(0)}"')
109 
110             # Remove all whitespace in the expression
111             expression = re.sub(r'\s', '', expression)
112 
113             # Wrap all exponents and transposition powers with {}
114             expression = re.sub(r'(?<=\^)(-?\d+|T)(?=[^}]|$)', r'{\g<0>}', expression)
115 
116             # Replace all subtractions with additions, multiplied by -1
117             expression = re.sub(r'(?<=.)-(?=[A-Z])', '+-1', expression)
118 
119             # Replace a possible leading minus sign with -1
120             expression = re.sub(r'^-(?=[A-Z])', '-1', expression)
121 
122             # Change all transposition exponents into lowercase
123             expression = expression.replace('^{T}', 't')
124 
125             # Split the expression into groups to be multiplied, and then we add those groups at the end
126             # We also have to filter out the empty strings to reduce errors
127             multiplication_groups = [x for x in expression.split('+') if x != '']
128 
129             # Start with the 0 matrix and add each group on
130             matrix_sum: MatrixType = np.array([[0., 0.], [0., 0.]])
131 
132             for group in multiplication_groups:
133                 # Generate a list of tuples, each representing a matrix
134                 # These tuples are (the multiplier, the matrix (with optional
135                 # 't' at the end to indicate a transpose), the exponent)
136                 string_matrices: list[tuple[str, str, str]]
137 
138                 # The generate tuple is (multiplier, matrix, full exponent, stripped exponent)
139                 # The full exponent contains ^{}, so we ignore it
140                 # The multiplier and exponent might be '', so we have to set them to '1'
141                 string_matrices = [(t[0] if t[0] != '' else '1', t[1], t[3] if t[3] != '' else '1')
142                                    for t in re.findall(r'(-?\d*\.?\d*)([A-Z]t?|rot\(\d+\))(\^{(-?\d+|T)})?', group)]
143 
144                 # This list is a list of tuple, where each tuple is (a float multiplier,
145                 # the matrix (gotten from the wrapper's __getitem__()), the integer power)
146                 matrices: list[tuple[float, MatrixType, int]]
147                 matrices = [(float(t[0]), self[t[1]], int(t[2])) for t in string_matrices]
148 
149                 # Process the matrices and make actual MatrixType objects
150                 processed_matrices: list[MatrixType] = [t[0] * np.linalg.matrix_power(t[1], t[2]) for t in matrices]
151 
152                 # Add this matrix product to the sum total
153                 matrix_sum += reduce(lambda m, n: m @ n, processed_matrices)
154 
155             return matrix_sum
```

I think the comments in the code speak for themselves, but we basically split the expression up into groups to be added, and then for each group, we multiply every matrix in that group to get its value, and then add all these values together at the end.

This code is objectively bad. At the time of writing, it's now quite old, so I can say that. This code has no real error handling, and line 127 introduces the glaring error that `'A++B'` is now a valid expression because we disregard empty strings. Not to mention the fact that the method is called `parse_expression()` but actually evaluates an expression. All these issues will be fixed in the future, but this was the first implementation of matrix evaluation, and it does the job decently well.

I then implemented several tests for this parsing.

```
# 60e0c713b244e097bab8ee0f71142b709fde1a8b
# tests/test_matrix_wrapper_parse_expression.py
```

```python
"""Test the MatrixWrapper parse_expression() method."""

import numpy as np
from numpy import linalg as la
import pytest
from lintrans.matrices import MatrixWrapper


@pytest.fixture
def wrapper() -> MatrixWrapper:
    """Return a new MatrixWrapper object with some preset values."""
    wrapper = MatrixWrapper()

    root_two_over_two = np.sqrt(2) / 2

    wrapper['A'] = np.array([[1, 2], [3, 4]])
    wrapper['B'] = np.array([[6, 4], [12, 9]])
    wrapper['C'] = np.array([[-1, -3], [4, -12]])
    wrapper['D'] = np.array([[13.2, 9.4], [-3.4, -1.8]])
    wrapper['E'] = np.array([
        [root_two_over_two, -1 * root_two_over_two],
        [root_two_over_two, root_two_over_two]
    ])
    wrapper['F'] = np.array([[-1, 0], [0, 1]])
    wrapper['G'] = np.array([[np.pi, np.e], [1729, 743.631]])

    return wrapper


def test_simple_matrix_addition(wrapper: MatrixWrapper) -> None:
    """Test simple addition and subtraction of two matrices."""

    # NOTE: We assert that all of these values are not None just to stop mypy complaining
    # These values will never actually be None because they're set in the wrapper() fixture
    # There's probably a better way  do this, because this method is a bit of a bdoge, but this works for now
    assert wrapper['A'] is not None and wrapper['B'] is not None and wrapper['C'] is not None and \
        wrapper['D'] is not None and wrapper['E'] is not None and wrapper['F'] is not None and \
        wrapper['G'] is not None

    assert (wrapper.parse_expression('A+B') == wrapper['A'] + wrapper['B']).all()
    assert (wrapper.parse_expression('E+F') == wrapper['E'] + wrapper['F']).all()
    assert (wrapper.parse_expression('G+D') == wrapper['G'] + wrapper['D']).all()
    assert (wrapper.parse_expression('C+C') == wrapper['C'] + wrapper['C']).all()
    assert (wrapper.parse_expression('D+A') == wrapper['D'] + wrapper['A']).all()
    assert (wrapper.parse_expression('B+C') == wrapper['B'] + wrapper['C']).all()


def test_simple_two_matrix_multiplication(wrapper: MatrixWrapper) -> None:
    """Test simple multiplication of two matrices."""
    assert wrapper['A'] is not None and wrapper['B'] is not None and wrapper['C'] is not None and \
           wrapper['D'] is not None and wrapper['E'] is not None and wrapper['F'] is not None and \
           wrapper['G'] is not None

    assert (wrapper.parse_expression('AB') == wrapper['A'] @ wrapper['B']).all()
    assert (wrapper.parse_expression('BA') == wrapper['B'] @ wrapper['A']).all()
    assert (wrapper.parse_expression('AC') == wrapper['A'] @ wrapper['C']).all()
    assert (wrapper.parse_expression('DA') == wrapper['D'] @ wrapper['A']).all()
    assert (wrapper.parse_expression('ED') == wrapper['E'] @ wrapper['D']).all()
    assert (wrapper.parse_expression('FD') == wrapper['F'] @ wrapper['D']).all()
    assert (wrapper.parse_expression('GA') == wrapper['G'] @ wrapper['A']).all()
    assert (wrapper.parse_expression('CF') == wrapper['C'] @ wrapper['F']).all()
    assert (wrapper.parse_expression('AG') == wrapper['A'] @ wrapper['G']).all()


def test_identity_multiplication(wrapper: MatrixWrapper) -> None:
    """Test that multiplying by the identity doesn't change the value of a matrix."""
    assert wrapper['A'] is not None and wrapper['B'] is not None and wrapper['C'] is not None and \
           wrapper['D'] is not None and wrapper['E'] is not None and wrapper['F'] is not None and \
           wrapper['G'] is not None

    assert (wrapper.parse_expression('I') == wrapper['I']).all()
    assert (wrapper.parse_expression('AI') == wrapper['A']).all()
    assert (wrapper.parse_expression('IA') == wrapper['A']).all()
```

```
74          assert (wrapper.parse_expression('GI') == wrapper['G']).all()
75          assert (wrapper.parse_expression('IG') == wrapper['G']).all()
76
77          assert (wrapper.parse_expression('EID') == wrapper['E'] @ wrapper['D']).all()
78          assert (wrapper.parse_expression('IED') == wrapper['E'] @ wrapper['D']).all()
79          assert (wrapper.parse_expression('EDI') == wrapper['E'] @ wrapper['D']).all()
80          assert (wrapper.parse_expression('IEIDI') == wrapper['E'] @ wrapper['D']).all()
81          assert (wrapper.parse_expression('EI^3D') == wrapper['E'] @ wrapper['D']).all()
82
83
84      def test_simple_three_matrix_multiplication(wrapper: MatrixWrapper) -> None:
85          """Test simple multiplication of two matrices."""
86          assert wrapper['A'] is not None and wrapper['B'] is not None and wrapper['C'] is not None and \
87                 wrapper['D'] is not None and wrapper['E'] is not None and wrapper['F'] is not None and \
88                 wrapper['G'] is not None
89
90          assert (wrapper.parse_expression('ABC') == wrapper['A'] @ wrapper['B'] @ wrapper['C']).all()
91          assert (wrapper.parse_expression('ACB') == wrapper['A'] @ wrapper['C'] @ wrapper['B']).all()
92          assert (wrapper.parse_expression('BAC') == wrapper['B'] @ wrapper['A'] @ wrapper['C']).all()
93          assert (wrapper.parse_expression('EFG') == wrapper['E'] @ wrapper['F'] @ wrapper['G']).all()
94          assert (wrapper.parse_expression('DAC') == wrapper['D'] @ wrapper['A'] @ wrapper['C']).all()
95          assert (wrapper.parse_expression('GAE') == wrapper['G'] @ wrapper['A'] @ wrapper['E']).all()
96          assert (wrapper.parse_expression('FAG') == wrapper['F'] @ wrapper['A'] @ wrapper['G']).all()
97          assert (wrapper.parse_expression('GAF') == wrapper['G'] @ wrapper['A'] @ wrapper['F']).all()
98
99
100     def test_matrix_inverses(wrapper: MatrixWrapper) -> None:
101         """Test the inverses of single matrices."""
102         assert wrapper['A'] is not None and wrapper['B'] is not None and wrapper['C'] is not None and \
103                wrapper['D'] is not None and wrapper['E'] is not None and wrapper['F'] is not None and \
104                wrapper['G'] is not None
105
106         assert (wrapper.parse_expression('A^{-1}') == la.inv(wrapper['A'])).all()
107         assert (wrapper.parse_expression('B^{-1}') == la.inv(wrapper['B'])).all()
108         assert (wrapper.parse_expression('C^{-1}') == la.inv(wrapper['C'])).all()
109         assert (wrapper.parse_expression('D^{-1}') == la.inv(wrapper['D'])).all()
110         assert (wrapper.parse_expression('E^{-1}') == la.inv(wrapper['E'])).all()
111         assert (wrapper.parse_expression('F^{-1}') == la.inv(wrapper['F'])).all()
112         assert (wrapper.parse_expression('G^{-1}') == la.inv(wrapper['G'])).all()
113
114         assert (wrapper.parse_expression('A^-1') == la.inv(wrapper['A'])).all()
115         assert (wrapper.parse_expression('B^-1') == la.inv(wrapper['B'])).all()
116         assert (wrapper.parse_expression('C^-1') == la.inv(wrapper['C'])).all()
117         assert (wrapper.parse_expression('D^-1') == la.inv(wrapper['D'])).all()
118         assert (wrapper.parse_expression('E^-1') == la.inv(wrapper['E'])).all()
119         assert (wrapper.parse_expression('F^-1') == la.inv(wrapper['F'])).all()
120         assert (wrapper.parse_expression('G^-1') == la.inv(wrapper['G'])).all()
121
122
123     def test_matrix_powers(wrapper: MatrixWrapper) -> None:
124         """Test that matrices can be raised to integer powers."""
125         assert wrapper['A'] is not None and wrapper['B'] is not None and wrapper['C'] is not None and \
126                wrapper['D'] is not None and wrapper['E'] is not None and wrapper['F'] is not None and \
127                wrapper['G'] is not None
128
129         assert (wrapper.parse_expression('A^2') == la.matrix_power(wrapper['A'], 2)).all()
130         assert (wrapper.parse_expression('B^4') == la.matrix_power(wrapper['B'], 4)).all()
131         assert (wrapper.parse_expression('C^{12}') == la.matrix_power(wrapper['C'], 12)).all()
132         assert (wrapper.parse_expression('D^12') == la.matrix_power(wrapper['D'], 12)).all()
133         assert (wrapper.parse_expression('E^8') == la.matrix_power(wrapper['E'], 8)).all()
134         assert (wrapper.parse_expression('F^{-6}') == la.matrix_power(wrapper['F'], -6)).all()
135         assert (wrapper.parse_expression('G^-2') == la.matrix_power(wrapper['G'], -2)).all()
```

These test lots of simple expressions, but don't test any more complicated expressions, nor do they test any validation, mostly because validation doesn't really exist at this point. `'A++B'` is still a valid expression and is equivalent to `'A+B'`.

3.1.3 Simple matrix expression validation

My next major step was to implement proper parsing, but I procrastinated for a while and first implemented proper validation.

```python
# 39b918651f60bc72bc19d2018075b24a6fc3af17
# src/lintrans/_parse/matrices.py

def compile_valid_expression_pattern() -> Pattern[str]:
    """Compile the single regular expression that will match a valid matrix expression."""
    digit_no_zero = '[123456789]'
    digits = '\\d+'
    integer_no_zero = '-?' + digit_no_zero + '(' + digits + ')?'
    real_number = f'({integer_no_zero}(\\.{digits})?|-?0?\\.{digits})'

    index_content = f'({integer_no_zero}|T)'
    index = f'(\\^\\{{{index_content}\\}}|\\^{index_content}|t)'
    matrix_identifier = f'([A-Z]|rot\\({real_number}\\))'
    matrix = '(' + real_number + '?' + matrix_identifier + index + '?)'
    expression = f'{matrix}+((\\+|-){matrix}+)*'

    return re.compile(expression)


# This is an expensive pattern to compile, so we compile it when this module is initialized
valid_expression_pattern = compile_valid_expression_pattern()


def validate_matrix_expression(expression: str) -> bool:
    """Validate the given matrix expression.

    This function simply checks the expression against a BNF schema. It is not
    aware of which matrices are actually defined in a wrapper. For an aware
    version of this function, use the MatrixWrapper().is_valid_expression() method.

    Here is the schema for a valid expression given in a version of BNF:

        expression        ::=  matrices { ( "+" | "-" ) matrices };
        matrices          ::=  matrix { matrix };
        matrix            ::=  [ real_number ] matrix_identifier [ index ];
        matrix_identifier ::=  "A" .. "Z" | "rot(" real_number ")";
        index             ::=  "^{" index_content "}" | "^" index_content | "t";
        index_content     ::=  integer_not_zero | "T";

        digit_no_zero     ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
        digit             ::=  "0" | digit_no_zero;
        digits            ::=  digit | digits digit;
        integer_not_zero  ::=  [ "-" ] digit_no_zero [ digits ];
        real_number       ::=  ( integer_not_zero [ "." digits ] | [ "-" ] [ "0" ] "." digits );

    :param str expression: The expression to be validated
    :returns bool: Whether the expression is valid according to the schema
    """
    match = valid_expression_pattern.match(expression)
    return expression == match.group(0) if match is not None else False
```

Here, I'm using a BNF schema to programmatically generate a regular expression. I use a function to generate this pattern and assign it to a variable when the module is initialized. This is because the pattern compilation is expensive and it's more efficient to compile the pattern once and then just use it in the `validate_matrix_expression()` function.

I also created a method `is_valid_expression()` in `MatrixWrapper`, which just validates a given expression. It uses the aforementioned `validate_matrix_expression()` and also checks that every matrix referenced in the expression is defined in the wrapper.

```python
# 39b918651f60bc72bc19d2018075b24a6fc3af17
# src/lintrans/matrices/wrapper.py
```

```
99          def is_valid_expression(self, expression: str) -> bool:
100             """Check if the given expression is valid, using the context of the wrapper.
101
102             This method calls _parse.validate_matrix_expression(), but also ensures
103             that all the matrices in the expression are defined in the wrapper.
104
105             :param str expression: The expression to validate
106             :returns bool: Whether the expression is valid according the schema
107             """
108             # Get rid of the transposes to check all capital letters
109             expression = re.sub(r'\^T', 't', expression)
110             expression = re.sub(r'\^{T}', 't', expression)
111
112             # Make sure all the referenced matrices are defined
113             for matrix in {x for x in expression if re.match('[A-Z]', x)}:
114                 if self[matrix] is None:
115                     return False
116
117             return _parse.validate_matrix_expression(expression)
```

I then implemented some simple tests to make sure the function works with valid and invalid expressions.

```
# a0fb029f7da995803c24ee36e7e8078e5621f676
# tests/_parse/test_parse_and_validate_expression.py

1   """Test the _parse.matrices module validation and parsing."""
2
3   import pytest
4   from lintrans._parse import validate_matrix_expression
5
6   valid_inputs: list[str] = [
7       'A', 'AB', '3A', '1.2A', '-3.4A', 'A^2', 'A^-1', 'A^{-1}',
8       'A^12', 'A^T', 'A^{5}', 'A^{T}', '4.3A^7', '9.2A^{18}',
9
10      'rot(45)', 'rot(12.5)', '3rot(90)',
11      'rot(135)^3', 'rot(51)^T', 'rot(-34)^-1',
12
13      'A+B', 'A+2B', '4.3A+9B', 'A^2+B^T', '3A^7+0.8B^{16}',
14      'A-B', '3A-4B', '3.2A^3-16.79B^T', '4.752A^{17}-3.32B^{36}',
15      'A--1B', '-A', '--1A'
16
17      '3A4B', 'A^TB', 'A^{T}B', '4A^6B^3',
18      '2A^{3}4B^5', '4rot(90)^3', 'rot(45)rot(13)',
19      'Arot(90)', 'AB^2', 'A^2B^2', '8.36A^T3.4B^12',
20
21      '3.5A^{4}5.6rot(19.2)^T-B^{-1}4.1C^5',
22  ]
23
24  invalid_inputs: list[str] = [
25      '', 'rot()', 'A^', 'A^1.2', 'A^{3.4}', '1,2A', 'ro(12)', '5', '12^2',
26      '^T', '^{12}', 'A^{13', 'A^3}', 'A^A', '^2', 'A--B', '--A'
27
28      'This is 100% a valid matrix expression, I swear'
29  ]
30
31
32  @pytest.mark.parametrize('inputs,output', [(valid_inputs, True), (invalid_inputs, False)])
33  def test_validate_matrix_expression(inputs: list[str], output: bool) -> None:
34      """Test the validate_matrix_expression() function."""
35      for inp in inputs:
36          assert validate_matrix_expression(inp) == output
```

Here, we test some valid data, some definitely invalid data, and some edge cases. At this stage, `'A--1B'` was considered a valid expression. This was a quirk of the validator at the time, but I fixed it later. This should obviously be an invalid expression, especially since `'A--B'` is considered invalid, but `'A--1B'` is valid.

The `@pytest.mark.parametrize` decorator on line 32 means that `pytest` will run one test for valid inputs, and then another test for invalid inputs, and these will count as different tests. This makes it easier to see which tests failed and then debug the app.

3.1.4 Parsing matrix expressions

Parsing is quite an interesting problem and something I didn't feel able to tackle head-on, so I wrote the unit tests first. I had a basic idea of what I wanted the parser to return, but no real idea of how to implement that. My unit tests looked like this:

```
# e9f7a81892278fe70684562052f330fb3a02bf9b
# tests/_parse/test_parse_and_validate_expression.py

40  expressions_and_parsed_expressions: list[tuple[str, MatrixParseList]] = [
41      # Simple expressions
42      ('A', [[('', 'A', '')]]),
43      ('A^2', [[('', 'A', '2')]]),
44      ('A^{2}', [[('', 'A', '2')]]),
45      ('3A', [[('3', 'A', '')]]),
46      ('1.4A^3', [[('1.4', 'A', '3')]]),
47
48      # Multiplications
49      ('4A^{3} 6B^2', [[('4', 'A', '3'), ('6', 'B', '2')]]),
50      ('4.2A^{T} 6.1B^-1', [[('4.2', 'A', 'T'), ('6.1', 'B', '-1')]]),
51      ('-1.2A^2 rot(45)^2', [[('-1.2', 'A', '2'), ('', 'rot(45)', '2')]]),
52      ('3.2A^T 4.5B^{5} 9.6rot(121.3)', [[('3.2', 'A', 'T'), ('4.5', 'B', '5'), ('9.6', 'rot(121.3)', '')]]),
53      ('-1.18A^{-2} 0.1B^{2} 9rot(34.6)^-1', [[('-1.18', 'A', '-2'), ('0.1', 'B', '2'), ('9', 'rot(34.6)', '-1')]]),
54
55      # Additions
56      ('A + B', [[('', 'A', '')], [('', 'B', '')]]),
57      ('A + B - C', [[('', 'A', '')], [('', 'B', '')], [('-1', 'C', '')]]),
58      ('2A^3 + 8B^T - 3C^-1', [[('2', 'A', '3')], [('8', 'B', 'T')], [('-3', 'C', '-1')]]),
59
60      # Additions with multiplication
61      ('2.14A^{3} 4.5rot(14.5)^-1 + 8B^T - 3C^-1', [[('2.14', 'A', '3'), ('4.5', 'rot(14.5)', '-1')],
62                                                    [('8', 'B', 'T')], [('-3', 'C', '-1')]]),
63      ('2.14A^{3} 4.5rot(14.5)^-1 + 8.5B^T 5.97C^4 - 3.14D^{-1} 6.7E^T',
64       [[('2.14', 'A', '3'), ('4.5', 'rot(14.5)', '-1')], [('8.5', 'B', 'T'), ('5.97', 'C', '4')],
65        [('-3.14', 'D', '-1'), ('6.7', 'E', 'T')]]),
66  ]
67
68
69  @pytest.mark.skip(reason='parse_matrix_expression() not implemented')
70  def test_parse_matrix_expression() -> None:
71      """Test the parse_matrix_expression() function."""
72      for expression, parsed_expression in expressions_and_parsed_expressions:
73          # Test it with and without whitespace
74          assert parse_matrix_expression(expression) == parsed_expression
75          assert parse_matrix_expression(expression.replace(' ', '')) == parsed_expression
```

I just had example inputs and what I expected as output. I also wanted the parser to ignore whitespace. The decorator on line 69 just skips the test because the parser wasn't implemented yet.

When implementing the parser, I first had to tighten up validation to remove anomalies like `'A--1B'` being valid. I did this by factoring out the optional minus signs from being part of a number, to being optionally in front of a number. This eliminated this kind of repetition and made `'A--1B'` invalid, as it should be.

```
# fd80d8d3b0e975e92dcc7c10f1f0f1276879f408
# src/lintrans/_parse/matrices.py

32  def compile_valid_expression_pattern() -> Pattern[str]:
33      """Compile the single regular expression that will match a valid matrix expression."""
34      digit_no_zero = '[123456789]'
35      digits = '\\d+'
```

```
36          integer_no_zero = digit_no_zero + '(' + digits + ')?'
37          real_number = f'({integer_no_zero}(\\.{digits})?|0?\\.{digits})'
38
39          index_content = f'(-?{integer_no_zero}|T)'
40          index = f'(\\^\\{{{index_content}\\}}|\\^{index_content}|t)'
41          matrix_identifier = f'([A-Z]|rot\\(-?{real_number}\\))'
42          matrix = '(' + real_number + '?' + matrix_identifier + index + '?)'
43          expression = f'-?{matrix}+((\\+|-){matrix}+)*'
44
45          return re.compile(expression)
```

The code can be a bit hard to read with all the RegEx stuff, but the BNF illustrates these changes nicely.

Compare the old version:

```
# 39b918651f60bc72bc19d2018075b24a6fc3af17
# src/lintrans/_parse/matrices.py

38  expression        ::=  matrices { ( "+" | "-" ) matrices };
39  matrices          ::=  matrix { matrix };
40  matrix            ::=  [ real_number ] matrix_identifier [ index ];
41  matrix_identifier ::=  "A" .. "Z" | "rot(" real_number ")";
42  index             ::=  "^{" index_content "}" | "^" index_content | "t";
43  index_content     ::=  integer_not_zero | "T";
44
45  digit_no_zero     ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
46  digit             ::=  "0" | digit_no_zero;
47  digits            ::=  digit | digits digit;
48  integer_not_zero  ::=  [ "-" ] digit_no_zero [ digits ];
49  real_number       ::=  ( integer_not_zero [ "." digits ] | [ "-" ] [ "0" ] "." digits );
```

to the new version:

```
# fd80d8d3b0e975e92dcc7c10f1f0f1276879f408
# src/lintrans/_parse/matrices.py

61  expression        ::=  [ "-" ] matrices { ( "+" | "-" ) matrices };
62  matrices          ::=  matrix { matrix };
63  matrix            ::=  [ real_number ] matrix_identifier [ index ];
64  matrix_identifier ::=  "A" .. "Z" | "rot(" [ "-" ] real_number ")";
65  index             ::=  "^{" index_content "}" | "^" index_content | "t";
66  index_content     ::=  [ "-" ] integer_not_zero | "T";
67
68  digit_no_zero     ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
69  digit             ::=  "0" | digit_no_zero;
70  digits            ::=  digit | digits digit;
71  integer_not_zero  ::=  digit_no_zero [ digits ];
72  real_number       ::=  ( integer_not_zero [ "." digits ] | [ "0" ] "." digits );
```

Then once I'd fixed the validation, I could implement the parser itself.

```
# fd80d8d3b0e975e92dcc7c10f1f0f1276879f408
# src/lintrans/_parse/matrices.py

86  def parse_matrix_expression(expression: str) -> MatrixParseList:
87      """Parse the matrix expression and return a list of results.
88
89      The return value is a list of results. This results list contains lists of tuples.
90      The top list is the expressions that should be added together, and each sublist
91      is expressions that should be multiplied together. These expressions to be
92      multiplied are tuples, where each tuple is (multiplier, matrix identifier, index).
93      The multiplier can be any real number, the matrix identifier is either a named
94      matrix or a new rotation matrix declared with 'rot()', and the index is an
95      integer or 'T' for transpose.
96
```

```
97          :param str expression: The expression to be parsed
98          :returns MatrixParseTuple: A list of results
99          """
100         # Remove all whitespace
101         expression = re.sub(r'\s', '', expression)
102
103         # Check if it's valid
104         if not validate_matrix_expression(expression):
105             raise MatrixParseError('Invalid expression')
106
107         # Wrap all exponents and transposition powers with {}
108         expression = re.sub(r'(?<=\^)(-?\d+|T)(?=[^}]|$)', r'{\g<0>}', expression)
109
110         # Remove any standalone minuses
111         expression = re.sub(r'-(?=[A-Z])', '-1', expression)
112
113         # Replace subtractions with additions
114         expression = re.sub(r'-(?=\d+\.?\d*([A-Z]|rot))', '+-', expression)
115
116         # Get rid of a potential leading + introduced by the last step
117         expression = re.sub(r'^\+', '', expression)
118
119         return [
120             [
121                 # The tuple returned by re.findall is (multiplier, matrix identifier, full index, stripped index),
122                 # so we have to remove the full index, which contains the {}
123                 (t[0], t[1], t[3])
124                 for t in re.findall(r'(-?\d+\.?\d*)?([A-Z]|rot\(-?\d+\.?\d*\))(\^{(-?\d+|T)})?', group)
125             ]
126             # We just split the expression by '+' to have separate groups
127             for group in expression.split('+')
128         ]
```

It works similarly to the old `MatrixWrapper.parse_expression()` method in §3.1.2 but with a powerful list comprehension at the end. It splits the expression up into groups and then uses some RegEx magic to find all the matrices in these groups as a tuple.

This method passes all the unit tests, as expected.

My next step was then to rewrite the evaluation to use this new parser, like so (method name and docstring removed):

```
# a453774bcdf824676461f9b9b441d7b94969ea55
# src/lintrans/matrices/wrapper.py
```

```
168         if not self.is_valid_expression(expression):
169             raise ValueError('The expression is invalid')
170
171         parsed_result = _parse.parse_matrix_expression(expression)
172         final_groups: list[list[MatrixType]] = []
173
174         for group in parsed_result:
175             f_group: list[MatrixType] = []
176
177             for matrix in group:
178                 if matrix[2] == 'T':
179                     m = self[matrix[1]]
180                     assert m is not None
181                     matrix_value = m.T
182                 else:
183                     matrix_value = np.linalg.matrix_power(self[matrix[1]],
184                                                           1 if (index := matrix[2]) == '' else int(index))
185
186                 matrix_value *= 1 if (multiplier := matrix[0]) == '' else float(multiplier)
187                 f_group.append(matrix_value)
188
189             final_groups.append(f_group)
190
191         return reduce(add, [reduce(matmul, group) for group in final_groups])
```

Here, we go through the list of tuples and evaluate the matrix represented by each tuple, putting this together in a list as we go. Then at the end, we simply reduce the sublists and then reduce these new matrices using a list comprehension in the `reduce()` call using `add` and `matmul` from the `operator` library. It's written in a functional programming style, and it passes all the previous tests.

3.2 Initial GUI

3.2.1 First basic GUI

The discrepancy in all the GUI code between `snake_case` and `camelCase` is because Qt5 was originally a C++ framework that was adapted into PyQt5 for Python. All the Qt API is in `camelCase`, but my Python code is in `snake_case`.

```
# 93ce763f7b993439fc0da89fad39456d8cc4b52c
# src/lintrans/gui/main_window.py

"""The module to provide the main window as a QMainWindow object."""

import sys

from PyQt5 import QtCore, QtGui, QtWidgets
from PyQt5.QtWidgets import QApplication, QHBoxLayout, QMainWindow, QVBoxLayout

from lintrans.matrices import MatrixWrapper


class LintransMainWindow(QMainWindow):
    """The class for the main window in the lintrans GUI."""

    def __init__(self):
        """Create the main window object, creating every widget in it."""
        super().__init__()

        self.matrix_wrapper = MatrixWrapper()

        self.setWindowTitle('Linear Transformations')
        self.setMinimumWidth(750)

        # === Create widgets

        # Left layout: the plot and input box

        # NOTE: This QGraphicsView is only temporary
        self.plot = QtWidgets.QGraphicsView(self)

        self.text_input_expression = QtWidgets.QLineEdit(self)
        self.text_input_expression.setPlaceholderText('Input matrix expression...')
        self.text_input_expression.textChanged.connect(self.update_render_buttons)

        # Right layout: all the buttons

        # Misc buttons

        self.button_create_polygon = QtWidgets.QPushButton(self)
        self.button_create_polygon.setText('Create polygon')
        # TODO: Implement create_polygon()
        # self.button_create_polygon.clicked.connect(self.create_polygon)
        self.button_create_polygon.setToolTip('Define a new polygon to view the transformation of')

        self.button_change_display_settings = QtWidgets.QPushButton(self)
        self.button_change_display_settings.setText('Change\ndisplay settings')
        # TODO: Implement change_display_settings()
        # self.button_change_display_settings.clicked.connect(self.change_display_settings)
        self.button_change_display_settings.setToolTip('Change which things are rendered on the plot')

        # Define new matrix buttons

```

```
52          self.label_define_new_matrix = QtWidgets.QLabel(self)
53          self.label_define_new_matrix.setText('Define a\nnew matrix')
54          self.label_define_new_matrix.setAlignment(QtCore.Qt.AlignCenter)
55
56          # TODO: Implement defining a new matrix visually, numerically, as a rotation, and as an expression
57
58          self.button_define_visually = QtWidgets.QPushButton(self)
59          self.button_define_visually.setText('Visually')
60          self.button_define_visually.setToolTip('Drag the basis vectors')
61
62          self.button_define_numerically = QtWidgets.QPushButton(self)
63          self.button_define_numerically.setText('Numerically')
64          self.button_define_numerically.setToolTip('Define a matrix just with numbers')
65
66          self.button_define_as_rotation = QtWidgets.QPushButton(self)
67          self.button_define_as_rotation.setText('As a rotation')
68          self.button_define_as_rotation.setToolTip('Define an angle to rotate by')
69
70          self.button_define_as_expression = QtWidgets.QPushButton(self)
71          self.button_define_as_expression.setText('As an expression')
72          self.button_define_as_expression.setToolTip('Define a matrix in terms of other matrices')
73
74          # Render buttons
75
76          self.button_render = QtWidgets.QPushButton(self)
77          self.button_render.setText('Render')
78          self.button_render.setEnabled(False)
79          self.button_render.clicked.connect(self.render_expression)
80          self.button_render.setToolTip('Render the expression<br><b>(Ctrl + Enter)</b>')
81
82          self.button_render_shortcut = QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Return'), self)
83          self.button_render_shortcut.activated.connect(self.button_render.click)
84
85          self.button_animate = QtWidgets.QPushButton(self)
86          self.button_animate.setText('Animate')
87          self.button_animate.setEnabled(False)
88          self.button_animate.clicked.connect(self.animate_expression)
89          self.button_animate.setToolTip('Animate the expression<br><b>(Ctrl + Shift + Enter)</b>')
90
91          self.button_animate_shortcut = QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Shift+Return'), self)
92          self.button_animate_shortcut.activated.connect(self.button_animate.click)
93
94          # === Arrange widgets
95
96          self.setContentsMargins(10, 10, 10, 10)
97
98          self.vlay_left = QVBoxLayout()
99          self.vlay_left.addWidget(self.plot)
100         self.vlay_left.addWidget(self.text_input_expression)
101
102         self.vlay_misc_buttons = QVBoxLayout()
103         self.vlay_misc_buttons.setSpacing(20)
104         self.vlay_misc_buttons.addWidget(self.button_create_polygon)
105         self.vlay_misc_buttons.addWidget(self.button_change_display_settings)
106
107         self.vlay_define_new_matrix = QVBoxLayout()
108         self.vlay_define_new_matrix.setSpacing(20)
109         self.vlay_define_new_matrix.addWidget(self.label_define_new_matrix)
110         self.vlay_define_new_matrix.addWidget(self.button_define_visually)
111         self.vlay_define_new_matrix.addWidget(self.button_define_numerically)
112         self.vlay_define_new_matrix.addWidget(self.button_define_as_rotation)
113         self.vlay_define_new_matrix.addWidget(self.button_define_as_expression)
114
115         self.vlay_render = QVBoxLayout()
116         self.vlay_render.setSpacing(20)
117         self.vlay_render.addWidget(self.button_animate)
118         self.vlay_render.addWidget(self.button_render)
119
120         self.vlay_right = QVBoxLayout()
121         self.vlay_right.setSpacing(50)
122         self.vlay_right.addLayout(self.vlay_misc_buttons)
123         self.vlay_right.addLayout(self.vlay_define_new_matrix)
124         self.vlay_right.addLayout(self.vlay_render)
```

```
125
126             self.hlay_all = QHBoxLayout()
127             self.hlay_all.setSpacing(15)
128             self.hlay_all.addLayout(self.vlay_left)
129             self.hlay_all.addLayout(self.vlay_right)
130
131             self.central_widget = QtWidgets.QWidget()
132             self.central_widget.setLayout(self.hlay_all)
133             self.setCentralWidget(self.central_widget)
134
135         def update_render_buttons(self) -> None:
136             """Enable or disable the render and animate buttons according to the validity of the matrix expression."""
137             valid = self.matrix_wrapper.is_valid_expression(self.text_input_expression.text())
138             self.button_render.setEnabled(valid)
139             self.button_animate.setEnabled(valid)
140
141         def render_expression(self) -> None:
142             """Render the expression in the input box, and then clear the box."""
143             # TODO: Render the expression
144             self.text_input_expression.setText('')
145
146         def animate_expression(self) -> None:
147             """Animate the expression in the input box, and then clear the box."""
148             # TODO: Animate the expression
149             self.text_input_expression.setText('')
150
151
152     def main() -> None:
153         """Run the GUI."""
154         app = QApplication(sys.argv)
155         window = LintransMainWindow()
156         window.show()
157         sys.exit(app.exec_())
158
159
160     if __name__ == '__main__':
161         main()
```

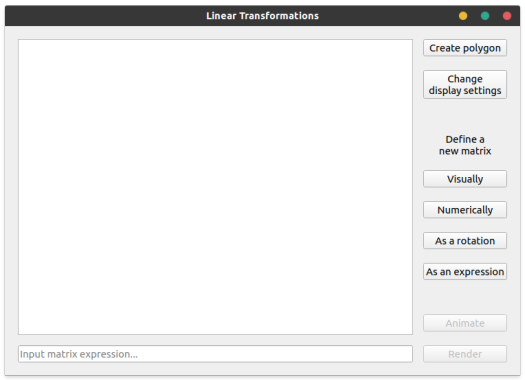


Figure 3.1: The first version of the GUI

A lot of the methods here don't have implementations yet, but they will. This version is just a very early prototype to get a rough draft of the GUI.

I create the widgets and layouts in the constructor as well as configuring all of them. The most important non-constructor method is `update_render_buttons()`. It gets called whenever the text in `text_input_expression` is changed. This happens because we connect it to the `textChanged` signal on line 32.

The big white box here will eventually be replaced with an actual viewport. This is just a prototype.

3.2.2 Numerical definition dialog

My next major addition was a dialog that would allow the user to define a matrix numerically.

```
# cedbd3ed126a1183f197c27adf6dabb4e5d301c7
# src/lintrans/gui/dialogs/define_new_matrix.py

1   """The module to provide dialogs for defining new matrices."""
2
3   from numpy import array
4   from PyQt5 import QtGui, QtWidgets
5   from PyQt5.QtWidgets import QDialog, QGridLayout, QHBoxLayout, QVBoxLayout
```

```
6
7   from lintrans.matrices import MatrixWrapper
8
9   ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
10
11
12  def is_float(string: str) -> bool:
13      """Check if a string is a float."""
14      try:
15          float(string)
16          return True
17      except ValueError:
18          return False
19
20
21  class DefineNumericallyDialog(QDialog):
22      """The dialog class that allows the user to define a new matrix numerically."""
23
24      def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
25          """Create the dialog, but don't run it yet.
26
27          :param matrix_wrapper: The MatrixWrapper that this dialog will mutate
28          :type matrix_wrapper: MatrixWrapper
29          """
30          super().__init__(*args, **kwargs)
31
32          self.matrix_wrapper = matrix_wrapper
33          self.setWindowTitle('Define a matrix')
34
35          # === Create the widgets
36
37          self.button_confirm = QtWidgets.QPushButton(self)
38          self.button_confirm.setText('Confirm')
39          self.button_confirm.setEnabled(False)
40          self.button_confirm.clicked.connect(self.confirm_matrix)
41          self.button_confirm.setToolTip('Confirm this as the new matrix<br><b>(Ctrl + Enter)</b>')
42
43          QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Return'), self).activated.connect(self.button_confirm.click)
44
45          self.button_cancel = QtWidgets.QPushButton(self)
46          self.button_cancel.setText('Cancel')
47          self.button_cancel.clicked.connect(self.close)
48          self.button_cancel.setToolTip('Cancel this definition<br><b>(Ctrl + Q)</b>')
49
50          QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Q'), self).activated.connect(self.button_cancel.click)
51
52          self.element_tl = QtWidgets.QLineEdit(self)
53          self.element_tl.textChanged.connect(self.update_confirm_button)
54
55          self.element_tr = QtWidgets.QLineEdit(self)
56          self.element_tr.textChanged.connect(self.update_confirm_button)
57
58          self.element_bl = QtWidgets.QLineEdit(self)
59          self.element_bl.textChanged.connect(self.update_confirm_button)
60
61          self.element_br = QtWidgets.QLineEdit(self)
62          self.element_br.textChanged.connect(self.update_confirm_button)
63
64          self.matrix_elements = (self.element_tl, self.element_tr, self.element_bl, self.element_br)
65
66          self.letter_combo_box = QtWidgets.QComboBox(self)
67
68          # Everything except I, because that's the identity
69          for letter in ALPHABET_NO_I:
70              self.letter_combo_box.addItem(letter)
71
72          self.letter_combo_box.activated.connect(self.load_matrix)
73
74          # === Arrange the widgets
75
76          self.setContentsMargins(10, 10, 10, 10)
77
78          self.grid_matrix = QGridLayout()
```

```
79              self.grid_matrix.setSpacing(20)
80              self.grid_matrix.addWidget(self.element_tl, 0, 0)
81              self.grid_matrix.addWidget(self.element_tr, 0, 1)
82              self.grid_matrix.addWidget(self.element_bl, 1, 0)
83              self.grid_matrix.addWidget(self.element_br, 1, 1)
84
85              self.hlay_buttons = QHBoxLayout()
86              self.hlay_buttons.setSpacing(20)
87              self.hlay_buttons.addWidget(self.button_cancel)
88              self.hlay_buttons.addWidget(self.button_confirm)
89
90              self.vlay_right = QVBoxLayout()
91              self.vlay_right.setSpacing(20)
92              self.vlay_right.addLayout(self.grid_matrix)
93              self.vlay_right.addLayout(self.hlay_buttons)
94
95              self.hlay_all = QHBoxLayout()
96              self.hlay_all.setSpacing(20)
97              self.hlay_all.addWidget(self.letter_combo_box)
98              self.hlay_all.addLayout(self.vlay_right)
99
100             self.setLayout(self.hlay_all)
101
102             # Finally, we load the default matrix A into the boxes
103             self.load_matrix(0)
104
105         def update_confirm_button(self) -> None:
106             """Enable the confirm button if there are numbers in every box."""
107             for elem in self.matrix_elements:
108                 if elem.text() == '' or not is_float(elem.text()):
109                     # If they're not all numbers, then we can't confirm it
110                     self.button_confirm.setEnabled(False)
111                     return
112
113             # If we didn't find anything invalid
114             self.button_confirm.setEnabled(True)
115
116         def load_matrix(self, index: int) -> None:
117             """If the selected matrix is defined, load it into the boxes."""
118             matrix = self.matrix_wrapper[ALPHABET_NO_I[index]]
119
120             if matrix is None:
121                 for elem in self.matrix_elements:
122                     elem.setText('')
123
124             else:
125                 self.element_tl.setText(str(matrix[0][0]))
126                 self.element_tr.setText(str(matrix[0][1]))
127                 self.element_bl.setText(str(matrix[1][0]))
128                 self.element_br.setText(str(matrix[1][1]))
129
130             self.update_confirm_button()
131
132         def confirm_matrix(self) -> None:
133             """Confirm the inputted matrix and assign it to the name."""
134             letter = self.letter_combo_box.currentText()
135             matrix = array([
136                 [float(self.element_tl.text()), float(self.element_tr.text())],
137                 [float(self.element_bl.text()), float(self.element_br.text())]
138             ])
139
140             self.matrix_wrapper[letter] = matrix
141             self.close()
```

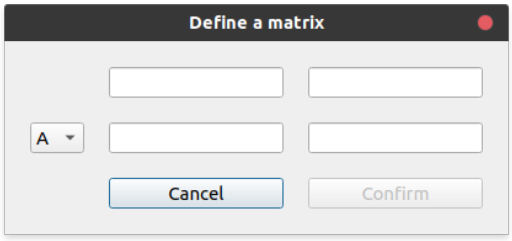


Figure 3.2: The first version of the numerical definition dialog

When I add more definition dialogs, I will factor out a superclass, but this is just a prototype to make sure it all works as intended.

Hopefully the methods are relatively self explanatory, but they're just utility methods to update the GUI when things are changed. We connect the `QLineEdit` widgets to the `update_confirm_button()` slot to make sure the confirm button is always up to date.

The `confirm_matrix()` method just updates the instance's matrix wrapper with the new matrix. We pass a reference to the `LintransMainWindow` instance's matrix wrapper when we open the dialog, so we're just updating the referenced object directly.

In the `LintransMainWindow` class, we're just connecting a lambda slot to the button so that it opens the dialog, as seen here:

```
# cedbd3ed126a1183f197c27adf6dabb4e5d301c7
# src/lintrans/gui/main_window.py

66  self.button_define_numerically.clicked.connect(
67      lambda: DefineNumericallyDialog(self.matrix_wrapper, self).exec()
68  )
```

3.2.3 More definition dialogs

I then factored out the constructor into a `DefineDialog` superclass so that I could easily create other definition dialogs.

```
# 5d04fb7233a03d0cd8fa0768f6387c6678da9df3
# src/lintrans/gui/dialogs/define_new_matrix.py

22  class DefineDialog(QDialog):
23      """A superclass for definitions dialogs."""
24
25      def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
26          """Create the dialog, but don't run it yet.
27
28          :param matrix_wrapper: The MatrixWrapper that this dialog will mutate
29          :type matrix_wrapper: MatrixWrapper
30          """
31          super().__init__(*args, **kwargs)
32
33          self.matrix_wrapper = matrix_wrapper
34          self.setWindowTitle('Define a matrix')
35
36          # === Create the widgets
37
38          self.button_confirm = QtWidgets.QPushButton(self)
39          self.button_confirm.setText('Confirm')
40          self.button_confirm.setEnabled(False)
41          self.button_confirm.clicked.connect(self.confirm_matrix)
42          self.button_confirm.setToolTip('Confirm this as the new matrix<br><b>(Ctrl + Enter)</b>')
43          QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self.button_confirm.click)
44
45          self.button_cancel = QtWidgets.QPushButton(self)
46          self.button_cancel.setText('Cancel')
47          self.button_cancel.clicked.connect(self.close)
48          self.button_cancel.setToolTip('Cancel this definition<br><b>(Ctrl + Q)</b>')
49          QShortcut(QKeySequence('Ctrl+Q'), self).activated.connect(self.button_cancel.click)
50
51          self.label_equals = QtWidgets.QLabel()
```

```
52          self.label_equals.setText('=')
53
54          self.letter_combo_box = QtWidgets.QComboBox(self)
55
56          # Everything except I, because that's the identity
57          for letter in ALPHABET_NO_I:
58              self.letter_combo_box.addItem(letter)
59
60          self.letter_combo_box.activated.connect(self.load_matrix)
```

This superclass just has a constructor that subclasses can use. When I added the `DefineAsARotationDialog` class, I also moved the cancel and confirm buttons into the constructor and added abstract methods that all dialog subclasses must implement.

```
# 0d534c35c6a4451e317d41a0d2b3ecb17827b45f
# src/lintrans/gui/dialogs/define_new_matrix.py

61          # === Arrange the widgets
62
63          self.setContentsMargins(10, 10, 10, 10)
64
65          self.horizontal_spacer = QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum)
66
67          self.hlay_buttons = QHBoxLayout()
68          self.hlay_buttons.setSpacing(20)
69          self.hlay_buttons.addItem(self.horizontal_spacer)
70          self.hlay_buttons.addWidget(self.button_cancel)
71          self.hlay_buttons.addWidget(self.button_confirm)
72
73      @property
74      def selected_letter(self) -> str:
75          """The letter currently selected in the combo box."""
76          return self.letter_combo_box.currentText()
77
78      @abc.abstractmethod
79      def update_confirm_button(self) -> None:
80          """Enable the confirm button if it should be enabled."""
81          ...
82
83      @abc.abstractmethod
84      def confirm_matrix(self) -> None:
85          """Confirm the inputted matrix and assign it.
86
87          This should mutate self.matrix_wrapper and then call self.accept().
88          """
89          ...
```

I then added the class for the rotation definition dialog.

```
# 0d534c35c6a4451e317d41a0d2b3ecb17827b45f
# src/lintrans/gui/dialogs/define_new_matrix.py

182 class DefineAsARotationDialog(DefineDialog):
183     """The dialog that allows the user to define a new matrix as a rotation."""
184
185     def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
186         """Create the dialog, but don't run it yet."""
187         super().__init__(matrix_wrapper, *args, **kwargs)
188
189         # === Create the widgets
190
191         self.label_equals.setText('= rot(')
192
193         self.text_angle = QtWidgets.QLineEdit(self)
194         self.text_angle.setPlaceholderText('angle')
195         self.text_angle.textChanged.connect(self.update_confirm_button)
196
197         self.label_close_paren = QtWidgets.QLabel(self)
```

```
198             self.label_close_paren.setText(')')
199
200             self.checkbox_radians = QtWidgets.QCheckBox(self)
201             self.checkbox_radians.setText('Radians')
202
203             # === Arrange the widgets
204
205             self.hlay_checkbox_and_buttons = QHBoxLayout()
206             self.hlay_checkbox_and_buttons.setSpacing(20)
207             self.hlay_checkbox_and_buttons.addWidget(self.checkbox_radians)
208             self.hlay_checkbox_and_buttons.addItem(self.horizontal_spacer)
209             self.hlay_checkbox_and_buttons.addLayout(self.hlay_buttons)
210
211             self.hlay_definition = QHBoxLayout()
212             self.hlay_definition.addWidget(self.letter_combo_box)
213             self.hlay_definition.addWidget(self.label_equals)
214             self.hlay_definition.addWidget(self.text_angle)
215             self.hlay_definition.addWidget(self.label_close_paren)
216
217             self.vlay_all = QVBoxLayout()
218             self.vlay_all.setSpacing(20)
219             self.vlay_all.addLayout(self.hlay_definition)
220             self.vlay_all.addLayout(self.hlay_checkbox_and_buttons)
221
222             self.setLayout(self.vlay_all)
223
224         def update_confirm_button(self) -> None:
225             """Enable the confirm button if there is a valid float in the angle box."""
226             self.button_confirm.setEnabled(is_float(self.text_angle.text()))
227
228         def confirm_matrix(self) -> None:
229             """Confirm the inputted matrix and assign it."""
230             self.matrix_wrapper[self.selected_letter] = create_rotation_matrix(
231                 float(self.text_angle.text()),
232                 degrees=not self.checkbox_radians.isChecked()
233             )
234             self.accept()
```

This dialog class just overrides the abstract methods of the superclass with its own implementations. This will be the pattern that all of the definition dialogs will follow.

It has a checkbox for radians, since this is supported in `create_rotation_matrix()`, but the textbox only supports numbers, so the user would have to calculate some multiple of π and paste in several decimal places. I expect people to only use degrees, because these are easier to use.

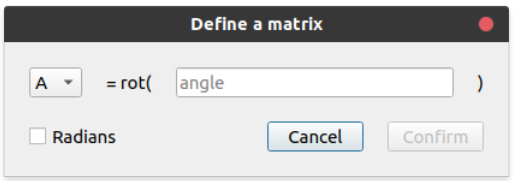


Figure 3.3: The first version of the rotation definition dialog

Additionally, I created a helper method in `LintransMainWindow`. Rather than connecting the `clicked` signal of the buttons to lambdas that instantiate an instance of the `DefineDialog` subclass and call `.exec()` on it, I now connect the `clicked` signal of the buttons to lambdas that call `self.dialog_define_matrix()` with the specific subclass.

```
# 6269e04d453df7be2d2f9c7ee176e83406ccc139
# src/lintrans/gui/main_window.py

170         def dialog_define_matrix(self, dialog_class: Type[DefineDialog]) -> None:
171             """Open a generic definition dialog to define a new matrix.
172
173             The class for the desired dialog is passed as an argument. We create an
174             instance of this class and the dialog is opened asynchronously and modally
175             (meaning it blocks interaction with the main window) with the proper method
176             connected to the ``dialog.finished`` slot.
177
178             .. note::
```

```
179             ``dialog_class`` must subclass :class:`lintrans.gui.dialogs.define_new_matrix.DefineDialog`.
180
181         :param dialog_class: The dialog class to instantiate
182         :type dialog_class: Type[lintrans.gui.dialogs.define_new_matrix.DefineDialog]
183         """
184         # We create a dialog with a deepcopy of the current matrix_wrapper
185         # This avoids the dialog mutating this one
186         dialog = dialog_class(deepcopy(self.matrix_wrapper), self)
187
188         # .open() is asynchronous and doesn't spawn a new event loop, but the dialog is still modal (blocking)
189         dialog.open()
190
191         # So we have to use the finished slot to call a method when the user accepts the dialog
192         # If the user rejects the dialog, this matrix_wrapper will be the same as the current one, because we copied
            ↪ it
193         # So we don't care, we just assign the wrapper anyway
194         dialog.finished.connect(lambda: self._assign_matrix_wrapper(dialog.matrix_wrapper))
195
196     def _assign_matrix_wrapper(self, matrix_wrapper: MatrixWrapper) -> None:
197         """Assign a new value to self.matrix_wrapper.
198
199         This is a little utility function that only exists because a lambda
200         callback can't directly assign a value to a class attribute.
201
202         :param matrix_wrapper: The new value of the matrix wrapper to assign
203         :type matrix_wrapper: MatrixWrapper
204         """
205         self.matrix_wrapper = matrix_wrapper
```

I also then implemented a simple `DefineAsAnExpressionDialog`, which evaluates a given expression in the current `MatrixWrapper` context and assigns the result to the given matrix name.

```
# d5f930e15c3c8798d4990486532da46e926a6cb9
# src/lintrans/gui/dialogs/define_new_matrix.py

241 class DefineAsAnExpressionDialog(DefineDialog):
242     """The dialog that allows the user to define a matrix as an expression."""
243
244     def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
245         """Create the dialog, but don't run it yet."""
246         super().__init__(matrix_wrapper, *args, **kwargs)
247
248         self.setMinimumWidth(450)
249
250         # === Create the widgets
251
252         self.text_box_expression = QtWidgets.QLineEdit(self)
253         self.text_box_expression.setPlaceholderText('Enter matrix expression...')
254         self.text_box_expression.textChanged.connect(self.update_confirm_button)
255
256         # === Arrange the widgets
257
258         self.hlay_definition.addWidget(self.text_box_expression)
259
260         self.vlay_all = QVBoxLayout()
261         self.vlay_all.setSpacing(20)
262         self.vlay_all.addLayout(self.hlay_definition)
263         self.vlay_all.addLayout(self.hlay_buttons)
264
265         self.setLayout(self.vlay_all)
266
267     def update_confirm_button(self) -> None:
268         """Enable the confirm button if the expression is valid."""
269         self.button_confirm.setEnabled(
270             self.matrix_wrapper.is_valid_expression(self.text_box_expression.text())
271         )
272
273     def confirm_matrix(self) -> None:
274         """Evaluate the matrix expression and assign its value to the chosen matrix."""
275         self.matrix_wrapper[self.selected_letter] = \
```

```
276                 self.matrix_wrapper.evaluate_expression(self.text_box_expression.text())
277             self.accept()
```

My next dialog that I wanted to implement was a visual definition dialog, which would allow the user to drag around the basis vectors to define a transformation. However, I would first need to create the `lintrans.gui.plots` package to allow for actually visualizing matrices and transformations.

3.3 Visualizing matrices

3.3.1 Asking strangers on the internet for help

After creating most of the GUI skeleton, I wanted to build the viewport. Unfortunately, I had no idea what I was doing.

While looking through the PyQt5 docs, I found a pretty comprehensive explanation of the Qt5 'Graphics View Framework'[14], which seemed pretty good, but not really what I was looking for. I wanted a way to easily draw lots of straight, parallel lines. This framework seemed more focussed on manipulating objects on a canvas, almost like sprites. I knew of a different Python library called `matplotlib`, which has various backends available. I learned that it could be embedded in a standard PyQt5 GUI, so I started doing some research.

I didn't get very far with `matplotlib`. I hadn't used it much before and it's designed for visualizing data. It can draw manually defined straight lines on a canvas, but that's not what it's designed for and it's not very good at it. Thankfully, my horrific `matplotlib` code has been lost to time. I used the `Qt5Agg` backend from `matplotlib` to create a custom PyQt5 widget for the GUI and I could graph randomly generated data with it after following a tutorial[13].

I realised that I wasn't going to get very far with `matplotlib`, but I didn't know what else to do. I couldn't find any relevant examples on the internet, so I decided to post a question on a forum myself. I'd had experience with StackOverflow and its unfriendly community before, so I decided to ask the `r/learnpython` subreddit[3].

I only got one response, but it was incredibly helpful. The person told me that if I couldn't find an easy way to do what I wanted, I could write a custom PyQt5 widget. I knew this was possible with a class that just inherited from `QWidget`, but had no idea how to actually make something useful. Thankfully, this person provided a link to a GitLab repository of theirs, where they had multiple examples of custom widgets with PyQt5[4].

When looking through this repo, I found out how to draw on a widget like a simple canvas. All I have to do is override the `paintEvent()` method and use a `QPainter` object to draw on the widget. I used this knowledge to start creating the actual viewport for the GUI, starting with the background axes.

3.3.2 Creating the plots package

Initially, the `lintrans.gui.plots` package just has some classes for widgets. `TransformationPlotWidget` acts as a base class and then `ViewTransformationWidget` acts as a wrapper. I will expand this class in the future.

```
# 4af63072b383dc9cef9adbb8900323aa007e7f26
# src/lintrans/gui/plots/plot_widget.py

"""This module provides the basic classes for plotting transformations."""

from __future__ import annotations

from PyQt5.QtCore import Qt
```

```python
from PyQt5.QtGui import QColor, QPainter, QPaintEvent, QPen
from PyQt5.QtWidgets import QWidget


class TransformationPlotWidget(QWidget):
    """An abstract superclass for plot widgets.

    This class provides a background (untransformed) plane, and all the backend
    details for a Qt application, but does not provide useful functionality. To
    be useful, this class must be subclassed and behaviour must be implemented
    by the subclass.

    .. warning:: This class should never be directly instantiated, only subclassed.

    .. note::
       I would make this class have ``metaclass=abc.ABCMeta``, but I can't because it subclasses ``QWidget``,
       and a every superclass of a class must have the same metaclass, and ``QWidget`` is not an abstract class.
    """

    def __init__(self, *args, **kwargs):
        """Create the widget, passing ``*args`` and ``**kwargs`` to the superclass constructor (``QWidget``)."""
        super().__init__(*args, **kwargs)

        self.setAutoFillBackground(True)

        # Set the background to white
        palette = self.palette()
        palette.setColor(self.backgroundRole(), Qt.white)
        self.setPalette(palette)

        # Set the gird colour to grey and the axes colour to black
        self.grid_colour = QColor(128, 128, 128)
        self.axes_colour = QColor(0, 0, 0)

        self.grid_spacing: int = 50
        self.line_width: float = 0.4

    @property
    def w(self) -> int:
        """Return the width of the widget."""
        return self.size().width()

    @property
    def h(self) -> int:
        """Return the height of the widget."""
        return self.size().height()

    def paintEvent(self, e: QPaintEvent):
        """Handle a ``QPaintEvent`` by drawing the widget."""
        qp = QPainter()
        qp.begin(self)
        self.draw_widget(qp)
        qp.end()

    def draw_widget(self, qp: QPainter):
        """Draw the grid and axes in the widget."""
        qp.setRenderHint(QPainter.Antialiasing)
        qp.setBrush(Qt.NoBrush)

        # Draw the grid
        qp.setPen(QPen(self.grid_colour, self.line_width))

        # We draw the background grid, centered in the middle
        # We deliberately exclude the axes - these are drawn separately
        for x in range(self.w // 2 + self.grid_spacing, self.w, self.grid_spacing):
            qp.drawLine(x, 0, x, self.h)
            qp.drawLine(self.w - x, 0, self.w - x, self.h)

        for y in range(self.h // 2 + self.grid_spacing, self.h, self.grid_spacing):
            qp.drawLine(0, y, self.w, y)
            qp.drawLine(0, self.h - y, self.w, self.h - y)

        # Now draw the axes
```

```
79          qp.setPen(QPen(self.axes_colour, self.line_width))
80          qp.drawLine(self.w // 2, 0, self.w // 2, self.h)
81          qp.drawLine(0, self.h // 2, self.w, self.h // 2)
82
83
84  class ViewTransformationWidget(TransformationPlotWidget):
85      """This class is used to visualise matrices as transformations."""
86
87      def __init__(self, *args, **kwargs):
88          """Create the widget, passing ``*args`` and ``**kwargs`` to the superclass constructor."""
89          super().__init__(*args, **kwargs)
```

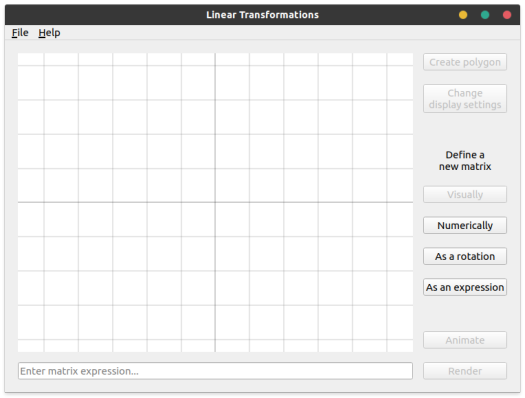


Figure 3.4: The GUI with background axes

The meat of this class is the `draw_widget()` method. Right now, this method only draws the background axes. My next step is to implement basis vector attributes and draw them in `draw_widget()`. After changing the the `plot` attribute in `LintransMainWindow` to an instance of `ViewTransformationWidget`, the plot was visible in the GUI.

I then refactored the code slightly to rename `draw_widget()` to `draw_background()` and then call it from the `paintEvent()` method in `ViewTransformationWidget`.

3.3.3 Implementing basis vectors

My first step in implementing basis vectors was to add some utility methods to convert between coordinate systems. The matrices are using Cartesian coordinates with $(0, 0)$ in the middle, positive x going to the right, and positive y going up. However, Qt5 is using standard computer graphics coordinates, with $(0, 0)$ in the top left, positive x going to the right, and positive y going down. I needed a way to convert Cartesian 'grid' coordinates to Qt5 'canvas' coordinates, so I wrote some little utility methods.

```
# 1fa7e1c61d61cb6aeff773b9698541f82fee39ea
# src/lintrans/gui/plots/plot_widget.py

45      @property
46      def origin(self) -> tuple[int, int]:
47          """Return the canvas coords of the origin."""
48          return self.width() // 2, self.height() // 2
49
50      def trans_x(self, x: float) -> int:
51          """Transform an x coordinate from grid coords to canvas coords."""
52          return int(self.origin[0] + x * self.grid_spacing)
53
54      def trans_y(self, y: float) -> int:
55          """Transform a y coordinate from grid coords to canvas coords."""
56          return int(self.origin[1] - y * self.grid_spacing)
57
58      def trans_coords(self, x: float, y: float) -> tuple[int, int]:
59          """Transform a coordinate in grid coords to canvas coords."""
60          return self.trans_x(x), self.trans_y(y)
```

Once I had a way to convert coordinates, I could add the basis vectors themselves. I did this by creating attributes for the points in the constructor and creating a `transform_by_matrix()` method to change these point attributes accordingly.

```
# 37e7c208a33d7cbbc8e0bb6c94cd889e2918c605
# src/lintrans/gui/plots/plot_widget.py
```

```
92  class ViewTransformationWidget(TransformationPlotWidget):
93      """This class is used to visualise matrices as transformations."""
94
95      def __init__(self, *args, **kwargs):
96          """Create the widget, passing ``*args`` and ``**kwargs`` to the superclass constructor."""
97          super().__init__(*args, **kwargs)
98
99          self.point_i: tuple[float, float] = (1., 0.)
100         self.point_j: tuple[float, float] = (0., 1.)
101
102         self.colour_i = QColor(37, 244, 15)
103         self.colour_j = QColor(8, 8, 216)
104
105         self.width_vector_line = 1
106         self.width_transformed_grid = 0.6
107
108     def transform_by_matrix(self, matrix: MatrixType) -> None:
109         """Transform the plane by the given matrix."""
110         self.point_i = (matrix[0][0], matrix[1][0])
111         self.point_j = (matrix[0][1], matrix[1][1])
112         self.update()
```

I also created a `draw_transformed_grid()` method which gets called in `paintEvent()`.

```
# 37e7c208a33d7cbbc8e0bb6c94cd889e2918c605
# src/lintrans/gui/plots/plot_widget.py

122     def draw_transformed_grid(self, painter: QPainter) -> None:
123         """Draw the transformed version of the grid, given by the unit vectors."""
124         # Draw the unit vectors
125         painter.setPen(QPen(self.colour_i, self.width_vector_line))
126         painter.drawLine(*self.origin, *self.trans_coords(*self.point_i))
127         painter.setPen(QPen(self.colour_j, self.width_vector_line))
128         painter.drawLine(*self.origin, *self.trans_coords(*self.point_j))
```

I then changed the `render_expression()` method in `LintransMainWindow` to call this new `transform_by_matrix()` method.

```
# 37e7c208a33d7cbbc8e0bb6c94cd889e2918c605
# src/lintrans/gui/main_window.py

229     def render_expression(self) -> None:
230         """Render the expression in the input box, and then clear the box."""
231         self.plot.transform_by_matrix(
232             self.matrix_wrapper.evaluate_expression(
233                 self.lineedit_expression_box.text()
234             )
235         )
```

Testing this new code shows that it works well.

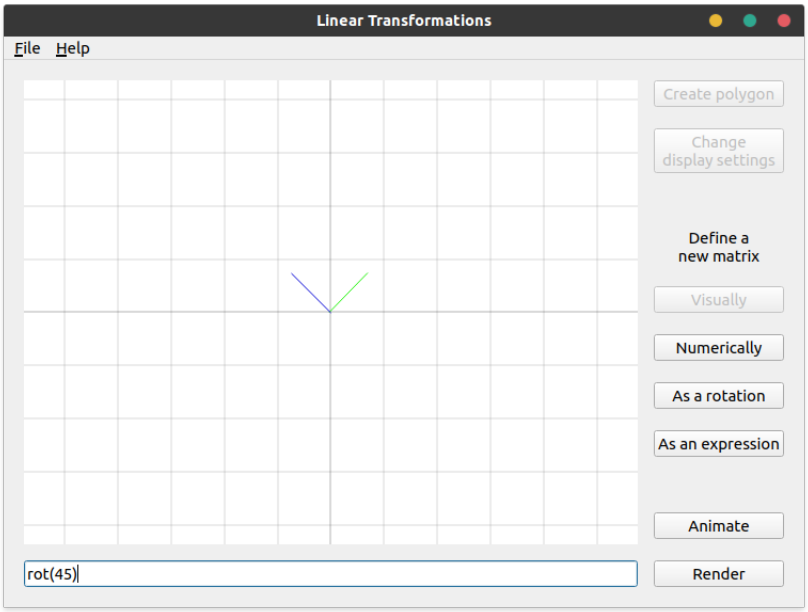


Figure 3.5: Basis vectors drawn for a 45° rotation

3.3.4 Drawing the transformed grid

After drawing the basis vectors, I wanted to draw the transformed version of the grid. I first created a `grid_corner()` utility method to return the grid coordinates of the top right corner of the canvas. This allows me to find the bounding box in which to draw the grid lines.

```
# 2ade98ac28d1c3f6691e4afa819142a3ab8e9fd9
# src/lintrans/gui/plots/plot_widget.py

64      def grid_corner(self) -> tuple[float, float]:
65          """Return the grid coords of the top right corner."""
66          return self.width() / (2 * self.grid_spacing), self.height() / (2 * self.grid_spacing)
```

I then created a `draw_parallel_lines()` method that would fill the bounding box with a set of lines parallel to a given vector with spacing defined by the intersection with a given point.

```
# 2ade98ac28d1c3f6691e4afa819142a3ab8e9fd9
# src/lintrans/gui/plots/plot_widget.py

126     def draw_parallel_lines(self, painter: QPainter, vector: tuple[float, float], point: tuple[float, float]) ->
        ↪ None:
127         """Draw a set of grid lines parallel to ``vector`` intersecting ``point``."""
128         max_x, max_y = self.grid_corner()
129         vector_x, vector_y = vector
130         point_x, point_y = point
131
132         if vector_x == 0:
133             painter.drawLine(self.trans_x(0), 0, self.trans_x(0), self.height())
134
135             for i in range(int(max_x / point_x)):
136                 painter.drawLine(
137                     self.trans_x((i + 1) * point_x),
138                     0,
139                     self.trans_x((i + 1) * point_x),
140                     self.height()
141                 )
142                 painter.drawLine(
143                     self.trans_x(-1 * (i + 1) * point_x),
```

```
                    0,
                    self.trans_x(-1 * (i + 1) * point_x),
                    self.height()
                )

        elif vector_y == 0:
            painter.drawLine(0, self.trans_y(0), self.width(), self.trans_y(0))

            for i in range(int(max_y / point_y)):
                painter.drawLine(
                    0,
                    self.trans_y((i + 1) * point_y),
                    self.width(),
                    self.trans_y((i + 1) * point_y)
                )
                painter.drawLine(
                    0,
                    self.trans_y(-1 * (i + 1) * point_y),
                    self.width(),
                    self.trans_y(-1 * (i + 1) * point_y)
                )
```

I then called this method from `draw_transformed_grid()`.

```
# 2ade98ac28d1c3f6691e4afa819142a3ab8e9fd9
# src/lintrans/gui/plots/plot_widget.py

    def draw_transformed_grid(self, painter: QPainter) -> None:
        """Draw the transformed version of the grid, given by the unit vectors."""
        # Draw the unit vectors
        painter.setPen(QPen(self.colour_i, self.width_vector_line))
        painter.drawLine(*self.origin, *self.trans_coords(*self.point_i))
        painter.setPen(QPen(self.colour_j, self.width_vector_line))
        painter.drawLine(*self.origin, *self.trans_coords(*self.point_j))

        # Draw all the parallel lines
        painter.setPen(QPen(self.colour_i, self.width_transformed_grid))
        self.draw_parallel_lines(painter, self.point_i, self.point_j)
        painter.setPen(QPen(self.colour_j, self.width_transformed_grid))
        self.draw_parallel_lines(painter, self.point_j, self.point_i)
```

This worked quite well when the matrix involved no rotation, as seen on the right, but this didn't work with rotation. When trying `'rot(45)'` for example, it looked the same as in Figure 3.5.

Also, the vectors aren't particularly clear. They'd be much better with arrowheads on their tips, but this is just a prototype. The arrowheads will come later.

My next step was to make the transformed grid lines work with rotations.

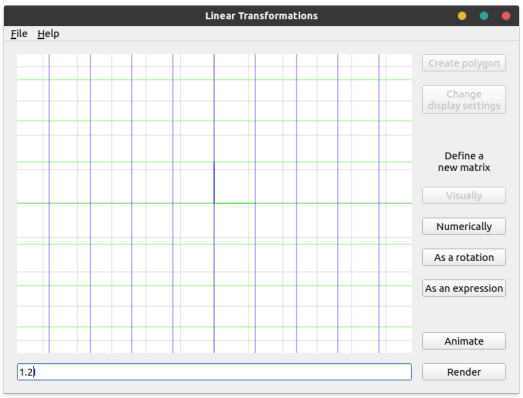


Figure 3.6: Parallel lines being drawn for matrix $1.2\mathbf{I}$

```
# 7dfe1e24729562501e2fd88a839dca6b653a3375
# src/lintrans/gui/plots/plot_widget.py

def draw_parallel_lines(self, painter: QPainter, vector: tuple[float, float], point: tuple[float, float]) -> None:
    """Draw a set of grid lines parallel to ``vector`` intersecting ``point``."""
    max_x, max_y = self.grid_corner()
    vector_x, vector_y = vector
    point_x, point_y = point
```

```
131
132         print(max_x, max_y, vector_x, vector_y, point_x, point_y)
133
134         # We want to use y = mx + c but m = y / x and if either of those are 0, then this
135         # equation is harder to work with, so we deal with these edge cases first
136         if abs(vector_x) < 1e-12 and abs(vector_y) < 1e-12:
137             # If both components of the vector are practically 0, then we can't render any grid lines
138             return
139
140         elif abs(vector_x) < 1e-12:
141             painter.drawLine(self.trans_x(0), 0, self.trans_x(0), self.height())
142
143             for i in range(abs(int(max_x / point_x))):
144                 painter.drawLine(
145                     self.trans_x((i + 1) * point_x),
146                     0,
147                     self.trans_x((i + 1) * point_x),
148                     self.height()
149                 )
150                 painter.drawLine(
151                     self.trans_x(-1 * (i + 1) * point_x),
152                     0,
153                     self.trans_x(-1 * (i + 1) * point_x),
154                     self.height()
155                 )
156
157         elif abs(vector_y) < 1e-12:
158             painter.drawLine(0, self.trans_y(0), self.width(), self.trans_y(0))
159
160             for i in range(abs(int(max_y / point_y))):
161                 painter.drawLine(
162                     0,
163                     self.trans_y((i + 1) * point_y),
164                     self.width(),
165                     self.trans_y((i + 1) * point_y)
166                 )
167                 painter.drawLine(
168                     0,
169                     self.trans_y(-1 * (i + 1) * point_y),
170                     self.width(),
171                     self.trans_y(-1 * (i + 1) * point_y)
172                 )
173
174         else:  # If the line is not horizontal or vertical, then we can use y = mx + c
175             m = vector_y / vector_x
176             c = point_y - m * point_x
177
178             # For c = 0
179             painter.drawLine(
180                 *self.trans_coords(
181                     -1 * max_x,
182                     m * -1 * max_x
183                 ),
184                 *self.trans_coords(
185                     max_x,
186                     m * max_x
187                 )
188             )
189
190             # Count up how many multiples of c we can have without wasting time rendering lines off screen
191             multiples_of_c: int = 0
192             ii: int = 1
193             while True:
194                 y1 = m * max_x + ii * c
195                 y2 = -1 * m * max_x + ii * c
196
197                 if y1 < max_y or y2 < max_y:
198                     multiples_of_c += 1
199                     ii += 1
200
201                 else:
202                     break
203
```

```
204             # Once we know how many lines we can draw, we just draw them all
205             for i in range(1, multiples_of_c + 1):
206                 painter.drawLine(
207                     *self.trans_coords(
208                         -1 * max_x,
209                         m * -1 * max_x + i * c
210                     ),
211                     *self.trans_coords(
212                         max_x,
213                         m * max_x + i * c
214                     )
215                 )
216                 painter.drawLine(
217                     *self.trans_coords(
218                         -1 * max_x,
219                         m * -1 * max_x - i * c
220                     ),
221                     *self.trans_coords(
222                         max_x,
223                         m * max_x - i * c
224                     )
225                 )
```

This code checks if x or y is zero[11] and if they're not, then we have to use the standard straight line equation $y = mx + c$ to create parallel lines. We find our value of m and then iterate through all the values of c that keep the line within the bounding box.

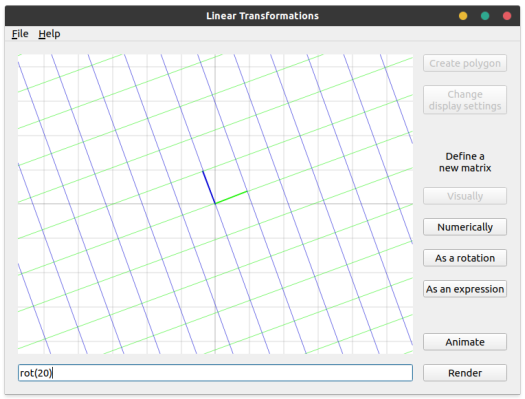


Figure 3.7: An example of a 20° rotation

There are some serious logical errors in this code. It works fine for things like `'3rot(45)'` or `'0.5rot(20)'`, but something like `'rot(115)'` will leave the program hanging indefinitely.

In fact, this code only works for rotations between 0° and 90°, and will hang forever when given a matrix like $\begin{pmatrix} 12 & 4 \\ -2 & 3 \end{pmatrix}$, because it's just not very good.

I will fix these issues in the future, but it works somewhat decently, so I decided to do animation next, because that sounded more fun.

3.3.5 Implementing animation

Now that I had a very crude renderer, I could create a method to animate a matrix. Eventually I want to be able to apply a given matrix to the currently rendered scene and animate between them. However, I wanted to start simple by animating from the identity to the given matrix.

```
# 829a130af5aee9819bf0269c03ecfb20bec1a108
# src/lintrans/gui/main_window.py

238     def animate_expression(self) -> None:
239         """Animate the expression in the input box, and then clear the box."""
240         self.button_render.setEnabled(False)
241         self.button_animate.setEnabled(False)
242
243         matrix = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())
244         matrix_move = matrix - self.matrix_wrapper['I']
245         steps: int = 100
246
247         for i in range(0, steps + 1):
```

[11] We actually check if they're less than 10^{-12} to allow for floating point errors

```
248             self.plot.visualize_matrix_transformation(
249                 self.matrix_wrapper['I'] + (i / steps) * matrix_move
250             )
251
252             self.update()
253             self.repaint()
254
255             time.sleep(0.01)
256
257         self.button_render.setEnabled(False)
258         self.button_animate.setEnabled(False)
```

This code creates the `matrix_move` variable and adds scaled versions of it to the identity matrix and renders that each frame. It's simple, but it works well for this simple use case. Unfortunately, it's very hard to show off an animation in a PDF, since all these images are static. The git commit hashes are included in the code snippets if you want to clone the repo[2], checkout this commit, and run it yourself if you want.

3.3.6 Preserving determinants

Ignoring the obvious flaw with not being able to render transformations with a more than 90° rotation, the animations don't respect determinants. When rotating 90°, the determinant changes during the animation, even though we're going from a determinant 1 matrix (the identity) to another determinant 1 matrix. This is because we're just moving each vector to its new position in a straight line. I want to animate in a way that smoothly transitions the determinant.

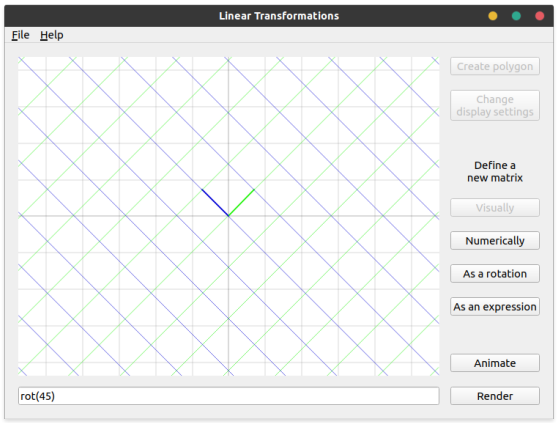


Figure 3.8: What we would expect halfway through a 90° rotation

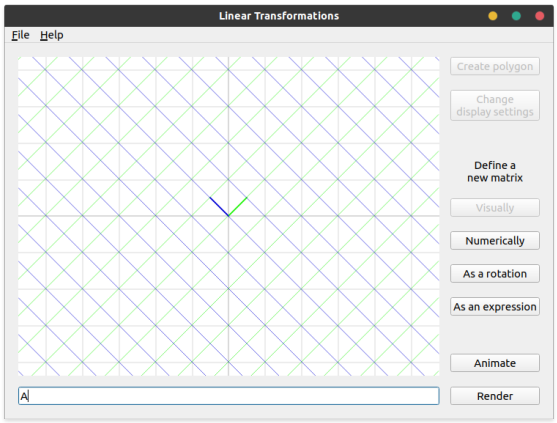


Figure 3.9: What we actually get halfway through a 90° rotation

In order to smoothly animate the determinant, I had to do some maths. I first defined the matrix $\mathbf{A}$ to be equivalent to the `matrix_move` variable from before - the target matrix minus the identity, scaled by the proportion. I then wanted to normalize $\mathbf{A}$ so that it had a determinant of 1 so that I could scale it up with the `proportion` variable through the animation.

I think I first tried just multiplying $\mathbf{A}$ by $\frac{1}{\det(\mathbf{A})}$ but that didn't work, so I googled it. I found a post[12] on ResearchGate about the topic, and thanks to a very helpful comment from Jeffrey L Stuart, I learned that for a 2×2 matrix $\mathbf{A}$ and a scalar c, $\det(c\mathbf{A}) = c^2 \det(\mathbf{A})$.

I wanted a c such that $\det(c\mathbf{A}) = 1$. Therefore $c = \frac{1}{\sqrt{|\det(\mathbf{A})|}}$. I then defined matrix $\mathbf{B}$ to be $c\mathbf{A}$.

Then I wanted to scale this normalized matrix $\mathbf{B}$ to have the same determinant as the target matrix $\mathbf{T}$ using some scalar d. We know that $\det(d\mathbf{B}) = d^2 \det(\mathbf{B}) = \det(\mathbf{T})$. We can just rearrange to find d

and get $d = \sqrt{\left|\frac{\det(\mathbf{T})}{\det(\mathbf{B})}\right|}$. But $\mathbf{B}$ is defined so that $\det(\mathbf{B}) = 1$, so we can get $d = \sqrt{|\det(\mathbf{T})|}$.

However, we want to scale this over time with our `proportion` variable p, so our final scalar $s = 1 + p\left(\sqrt{|\det(\mathbf{T})|} - 1\right)$. We define a matrix $\mathbf{C} = s\mathbf{B}$ and render $\mathbf{C}$ each frame. When in code form, this is the following:

```
# 6ff49450d8438ea2b2e7d2a97125dc518e648bc5
# src/lintrans/gui/main_window.py
```

```
245             # Get the target matrix and it's determinant
246             matrix_target = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())
247             det_target = linalg.det(matrix_target)
248 
249             identity = self.matrix_wrapper['I']
250             steps: int = 100
251 
252             for i in range(0, steps + 1):
253                 # This proportion is how far we are through the loop
254                 proportion = i / steps
255 
256                 # matrix_a is the identity plus some part of the target, scaled by the proportion
257                 # If we just used matrix_a, then things would animate, but the determinants would be weird
258                 matrix_a = identity + proportion * (matrix_target - identity)
259 
260                 # So to fix the determinant problem, we get the determinant of matrix_a and use it to normalise
261                 det_a = linalg.det(matrix_a)
262 
263                 # For a 2x2 matrix A and a scalar c, we know that det(cA) = c^2 det(A)
264                 # We want B = cA such that det(B) = 1, so then we can scale it with the animation
265                 # So we get c^2 det(A) = 1 => c = sqrt(1 / abs(det(A)))
266                 # Then we scale A down to get a determinant of 1, and call that matrix_b
267                 if det_a == 0:
268                     c = 0
269                 else:
270                     c = np.sqrt(1 / abs(det_a))
271 
272                 matrix_b = c * matrix_a
273 
274                 # matrix_c is the final matrix that we transform by
275                 # It's B, but we scale it up over time to have the target determinant
276 
277                 # We want some C = dB such that det(C) is some target determinant T
278                 # det(dB) = d^2 det(B) = T => d = sqrt(abs(T / det(B))
279                 # But we defined B to have det 1, so we can ignore it there
280 
281                 # We're also subtracting 1 and multiplying by the proportion and then adding one
282                 # This just scales the determinant along with the animation
283                 scalar = 1 + proportion * (np.sqrt(abs(det_target)) - 1)
284 
285                 matrix_c = scalar * matrix_b
286 
287                 self.plot.visualize_matrix_transformation(matrix_c)
288 
289                 self.repaint()
290                 time.sleep(0.01)
```

Unfortunately, the system I use to render matrices is still quite bad at its job. This makes it hard to test properly. But, transformations like `'2rot(90)'` work exactly as expected, which is very good.

3.4 Improving the GUI

3.4.1 Fixing rendering

Now that I had the basics of matrix visualization sorted, I wanted to make the GUI and UX better. My first step was overhauling the rendering code to make it actually work with rotations of more than 90°.

I narrowed down the issue with PyCharm's debugger and found that the loop in `VectorGridPlot.draw_parallel_lines()` was looping forever if it tried to doing anything outside of the top right quadrant. To fix this, I decided to instead delegate this task of drawing a set of oblique lines to a separate method, and work on that instead.

```
# cf05e09e5ebb6ea7a96db8660d0d8de6b946490a
# src/lintrans/gui/plots/classes.py
```

```
203             else:  # If the line is not horizontal or vertical, then we can use y = mx + c
204                 m = vector_y / vector_x
205                 c = point_y - m * point_x
206 
207                 # For c = 0
208                 painter.drawLine(
209                     *self.trans_coords(
210                         -1 * max_x,
211                         m * -1 * max_x
212                     ),
213                     *self.trans_coords(
214                         max_x,
215                         m * max_x
216                     )
217                 )
218 
219                 # We keep looping and increasing the multiple of c until we stop drawing lines on the canvas
220                 multiple_of_c = 1
221                 while self.draw_pair_of_oblique_lines(painter, m, multiple_of_c * c):
222                     multiple_of_c += 1
```

This separation of functionality made designing and debugging this part of the solution much easier. The `draw_pair_of_oblique_lines()` method looked like this:

```
# cf05e09e5ebb6ea7a96db8660d0d8de6b946490a
# src/lintrans/gui/plots/classes.py
```

```
224     def draw_pair_of_oblique_lines(self, painter: QPainter, m: float, c: float) -> bool:
225         """Draw a pair of oblique lines, using the equation y = mx + c.
226 
227         This method just calls :meth:`draw_oblique_line` with ``c`` and ``-c``,
228         and returns True if either call returned True.
229 
230         :param QPainter painter: The ``QPainter`` object to use for drawing the vectors and grid lines
231         :param float m: The gradient of the lines to draw
232         :param float c: The y-intercept of the lines to draw. We use the positive and negative versions
233         :returns bool: Whether we were able to draw any lines on the canvas
234         """
235         return any([
236             self.draw_oblique_line(painter, m, c),
237             self.draw_oblique_line(painter, m, -c)
238         ])
239 
240     def draw_oblique_line(self, painter: QPainter, m: float, c: float) -> bool:
241         """Draw an oblique line, using the equation y = mx + c.
242 
243         We only draw the part of the line that fits within the canvas, returning True if
244         we were able to draw a line within the boundaries, and False if we couldn't draw a line
245 
246         :param QPainter painter: The ``QPainter`` object to use for drawing the vectors and grid lines
```

```
247         :param float m: The gradient of the line to draw
248         :param float c: The y-intercept of the line to draw
249         :returns bool: Whether we were able to draw a line on the canvas
250         """
251         max_x, max_y = self.grid_corner()
252 
253         # These variable names are shortened for convenience
254         # myi is max_y_intersection, mmyi is minus_max_y_intersection, etc.
255         myi = (max_y - c) / m
256         mmyi = (-max_y - c) / m
257         mxi = max_x * m + c
258         mmxi = -max_x * m + c
259 
260         # The inner list here is a list of coords, or None
261         # If an intersection fits within the bounds, then we keep its coord,
262         # else it is None, and then gets discarded from the points list
263         # By the end, points is a list of two coords, or an empty list
264         points: list[tuple[float, float]] = [
265             x for x in [
266                 (myi, max_y) if -max_x < myi < max_x else None,
267                 (mmyi, -max_y) if -max_x < mmyi < max_x else None,
268                 (max_x, mxi) if -max_y < mxi < max_y else None,
269                 (-max_x, mmxi) if -max_y < mmxi < max_y else None
270             ] if x is not None
271         ]
272 
273         # If no intersections fit on the canvas
274         if len(points) < 2:
275             return False
276 
277         # If we can, then draw the line
278         else:
279             painter.drawLine(
280                 *self.trans_coords(*points[0]),
281                 *self.trans_coords(*points[1])
282             )
283             return True
```

To illustrate what this code is doing, I'll use a diagram.

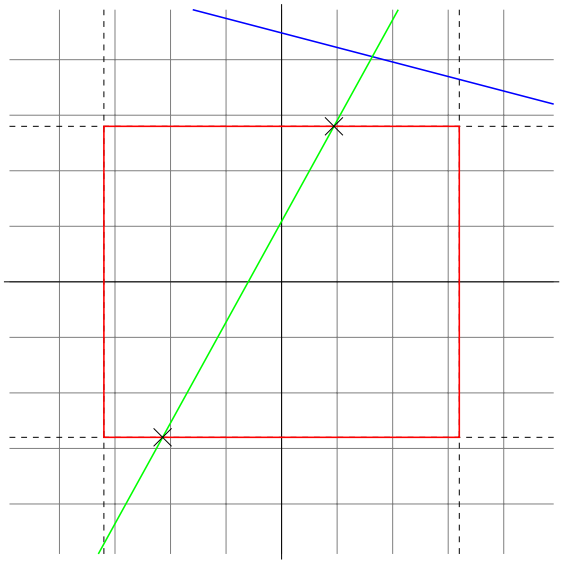


Figure 3.10: Two example lines and the viewport box

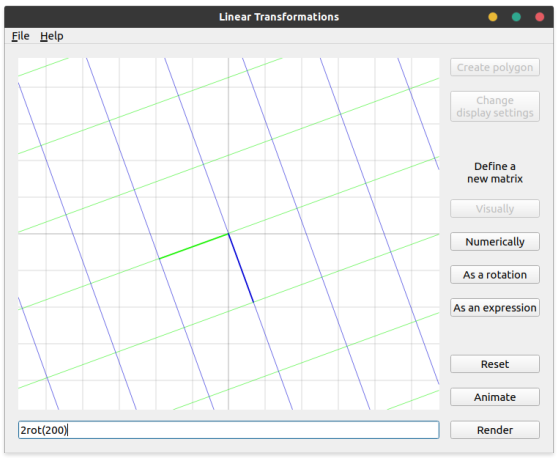


Figure 3.11: A demonstration of the new oblique lines system.

The red box represents the viewport of the GUI. The dashed lines represent the extensions of the red box. For a given line we want to draw, we first want to find where it intersects these orthogonal lines. Any oblique line will intersect each of these lines exactly once. This is what the `myi`, `mmyi`, `mxi`, and

`mmxi` variables represent. The value of `myi` is the x value where the line intersects the maximum y line, for example.

In the case of the blue line, all 4 intersection points are outside the bounds of the box, whereas the green line intersects with the box, as shown with the crosses. We use a list comprehension over a list of ternaries to get the `points` list. This list contains 0 or 2 coordinates, and we may or may not draw a line accordingly.

That's how the `draw_oblique_line()` method works, and the `draw_pair_of_oblique_lines()` method just calls it with positive and negative values of c.

3.4.2 Adding vector arrowheads

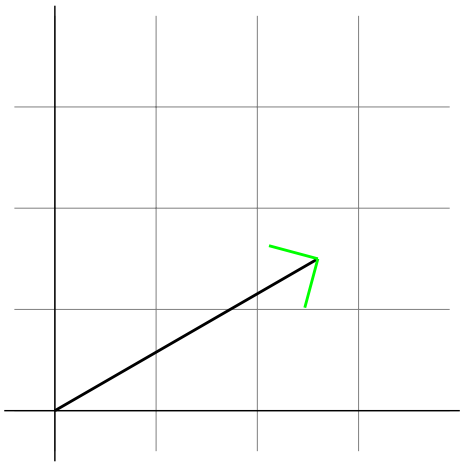


Figure 3.12: An example of a vector with the arrowheads highlighted in green

Now that I had a good renderer, I wanted to add arrowheads to the vectors to make them easier to see. They were already thicker than the gridlines, but adding arrowheads like in the 3blue1brown series would make them much easier to see. Unfortunately, I couldn't work out how to do this.

I wanted a function that would take a coordinate, treat it as a unit vector, and draw lines at 45° angles at the tip. This wasn't how I was conceptualising the problem at the time and because of that, I couldn't work out how to solve this problem. I could create this 45° lines in the top right quadrant, but none of my possible solutions worked for any arbitrary point.

So I started googling and found a very nice algorithm on `csharphelper.com`[23], which I adapted for Python.

```
# 5373b1ad8040f6726147cccea523c0570251cf67
# src/lintrans/gui/plots/widgets.py
```

```
52      def draw_arrowhead_away_from_origin(self, painter: QPainter, point: tuple[float, float]) -> None:
53          """Draw an arrowhead at ``point``, pointing away from the origin.
54
55          :param QPainter painter: The ``QPainter`` object to use to draw the arrowheads with
56          :param point: The point to draw the arrowhead at, given in grid coords
57          :type point: tuple[float, float]
58          """
59          # This algorithm was adapted from a C# algorithm found at
60          # http://csharphelper.com/blog/2014/12/draw-lines-with-arrowheads-in-c/
61
62          # Get the x and y coords of the point, and then normalize them
63          # We have to normalize them, or else the size of the arrowhead will
64          # scale with the distance of the point from the origin
65          x, y = point
66          nx = x / np.sqrt(x * x + y * y)
67          ny = y / np.sqrt(x * x + y * y)
68
69          # We choose a length and do some magic to find the steps in the x and y directions
70          length = 0.15
71          dx = length * (-nx - ny)
72          dy = length * (nx - ny)
73
74          # Then we just plot those lines
75          painter.drawLine(*self.trans_coords(x, y), *self.trans_coords(x + dx, y + dy))
76          painter.drawLine(*self.trans_coords(x, y), *self.trans_coords(x - dy, y + dx))
77
78      def draw_vector_arrowheads(self, painter: QPainter) -> None:
79          """Draw arrowheads at the tips of the basis vectors.
```

```
80
81          :param QPainter painter: The ``QPainter`` object to use to draw the arrowheads with
82          """
83          painter.setPen(QPen(self.colour_i, self.width_vector_line))
84          self.draw_arrowhead_away_from_origin(painter, self.point_i)
85          painter.setPen(QPen(self.colour_j, self.width_vector_line))
86          self.draw_arrowhead_away_from_origin(painter, self.point_j)
```

As the comments suggest, we get the x and y components of the normalised vector, and then do some magic with a chosen length and get some distance values, and then draw those lines. I don't really understand how this code works, but I'm happy that it does. All we have to do is call `draw_vector_arrowheads()` from `paintEvent()`.

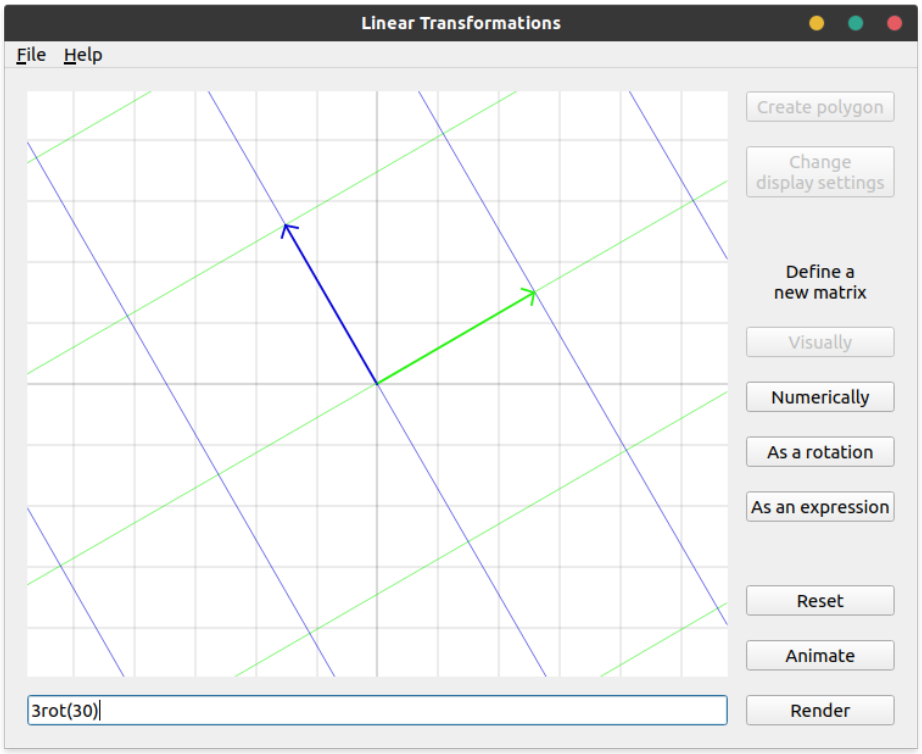


Figure 3.13: An example of the i and j vectors with arrowheads

3.4.3 Implementing zoom

The next thing I wanted to do was add the ability to zoom in and out of the viewport, and I wanted a button to reset the zoom level as well. I added a `default_grid_spacing` class attribute in `BackgroundPlot` and used that as the `grid_spacing` instance attribute in `__init__()`.

```
# d944e86e1d0fdc2c4be4d63479bc6bc3a31568ef
# src/lintrans/gui/plots/classes.py

27      default_grid_spacing: int = 50
28
29      def __init__(self, *args, **kwargs):
30          """Create the widget and setup backend stuff for rendering.
31
32          .. note:: ``*args`` and ``**kwargs`` are passed the superclass constructor (``QWidget``).
33          """
34          super().__init__(*args, **kwargs)
35
36          self.setAutoFillBackground(True)
```

```
37
38          # Set the background to white
39          palette = self.palette()
40          palette.setColor(self.backgroundRole(), Qt.white)
41          self.setPalette(palette)
42
43          # Set the gird colour to grey and the axes colour to black
44          self.colour_background_grid = QColor(128, 128, 128)
45          self.colour_background_axes = QColor(0, 0, 0)
46
47          self.grid_spacing = BackgroundPlot.default_grid_spacing
```

The reset button in `LintransMainWindow` simply sets `plot.grid_spacing` to the default.

To actually allow for zooming, I had to implement the `wheelEvent()` method in `BackgroundPlot` to listen for mouse wheel events. After reading through the docs for the `QWheelEvent` class[18], I learned how to handle this event.

```
# d944e86e1d0fdc2c4be4d63479bc6bc3a31568ef
# src/lintrans/gui/plots/classes.py

119     def wheelEvent(self, event: QWheelEvent) -> None:
120         """Handle a ``QWheelEvent`` by zooming in or our of the grid."""
121         # angleDelta() returns a number of units equal to 8 times the number of degrees rotated
122         degrees = event.angleDelta() / 8
123
124         if degrees is not None:
125             self.grid_spacing = max(1, self.grid_spacing + degrees.y())
126
127         event.accept()
128         self.update()
```

All we do is get the amount that the user scrolled and add that to the current spacing, taking the max with 1, which acts as a minimum grid spacing. We need to use `degrees.y()` on line 125 because Qt5 allows for mice that can scroll in the x and y directions, and we only want the y component. Line 127 marks the event as accepted so that the parent widget doesn't try to act on it.

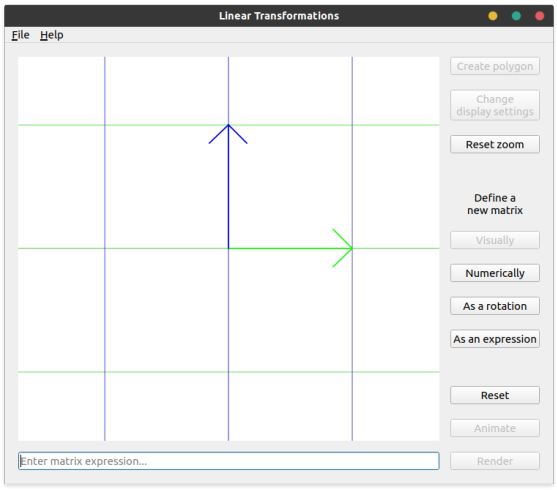


Figure 3.14: The GUI zoomed in a bit

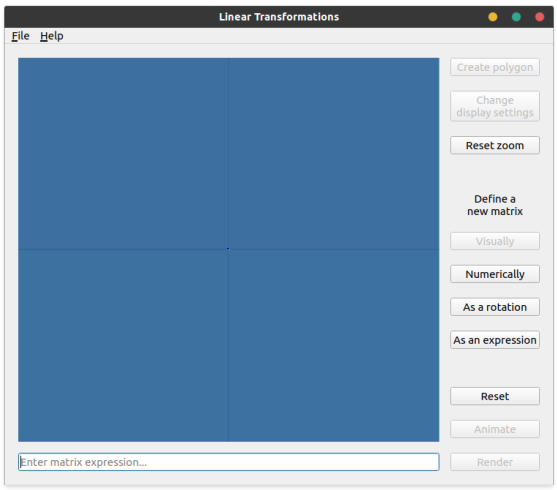


Figure 3.15: The GUI zoomed out as far as possible

There are two things I don't like here. Firstly, the minimum grid spacing is too small. The user can zoom out too far. Secondly, the arrowheads are too big in figure 3.14.

The first problem is minor and won't be fixed for quite a while, but I fixed the second problem quite quickly.

We want the arrowhead length to not just be 0.15, but to scale with the zoom level (the ratio between default grid spacing and current spacing).

This creates a slight issue when zoomed out all the way, because the arrowheads are then far larger than the vectors themselves, so we take the minimum of the scaled length and the vector length.

I factored out the default arrowhead length into the `arrowhead_length` instance attribute and initialize it in `__init__()`.

```
# 3d19a003368ae992ebb60049685bb04fde0836b5
# src/lintrans/gui/plots/widgets.py

68          vector_length = np.sqrt(x * x + y * y)
69          nx = x / vector_length
70          ny = y / vector_length
71
72          # We choose a length and find the steps in the x and y directions
73          length = min(
74              self.arrowhead_length * self.default_grid_spacing / self.grid_spacing,
75              vector_length
76          )
```

This code results in arrowheads that stay the same length unless the user is zoomed out basically as far as possible.

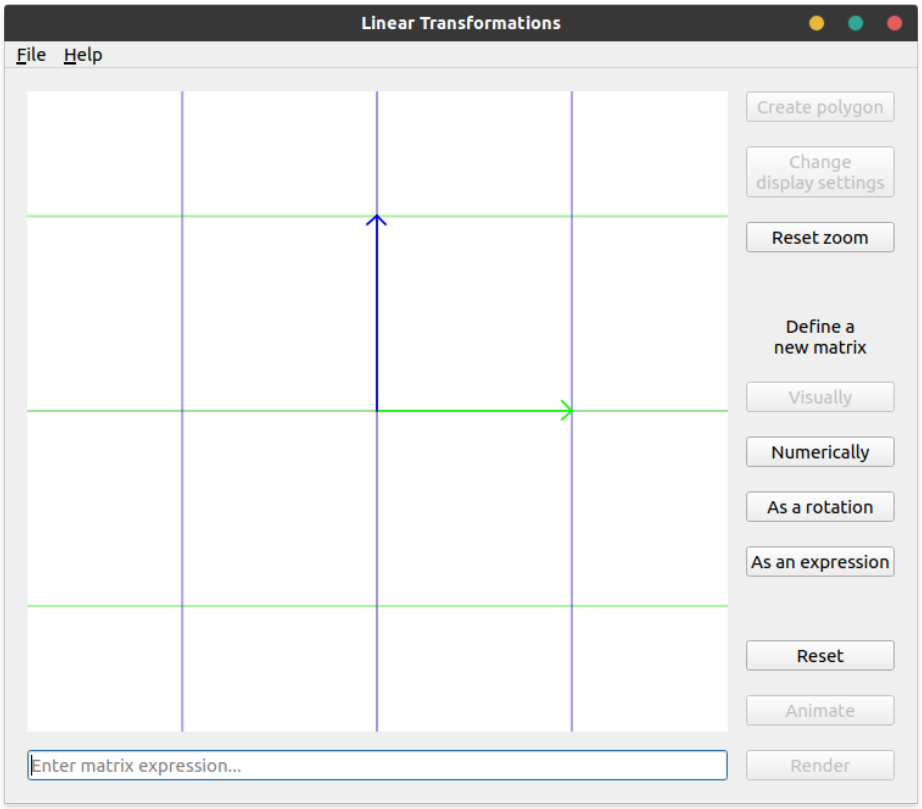


Figure 3.16: The arrowheads adjusted for zoom level

3.4.4 Animation blocks zooming

The biggest problem with this new zoom feature is that when animating between matrices, the user is unable to zoom. This is because when `LintransMainWindow.animate_expression()` is called, it uses

Python's standard library `time.sleep()` function to delay each frame, which prevents Qt from handling user interaction while we're animating. This was a problem.

I did some googling and found a helpful post on StackOverflow[9] that gave me a nice solution. The user `ekhumoro` used the functions `QApplication.processEvents()` and `QThread.msleep()` to solve the problem, and I used these functions in my own app, with much success.

After reading 'The Event System' in the Qt5 documentation[24], I learned that Qt5 uses an event loop, a lot like JavaScript. This means that events are scheduled to be executed on the next pass of the event loop. I also read the documentation for the `repaint()` and `update()` methods on the `QWidget` class[20, 21] and decided that it would be better to just queue a repaint by calling `update()` on the plot rather than immediately repaint with `repaint()`, and then call `QApplication.processEvents()` to process the pending events on the main thread. This is a nicer way of repainting, which reduces potential flickering issues, and using `QThread.msleep()` allows for asynchronous processing and therefore non-blocking animation.

3.4.5 Rank 1 transformations

The rank of a matrix is the dimension of its column space. This is the dimension of the span of its columns, which is to say the dimension of the output space. The rank of a matrix must be less than or equal to the dimension of the matrix, so we only need to worry about ranks 0, 1, and 2. There is only one rank 0 matrix, which is the **0** matrix itself. I've already covered this case by just not drawing any transformed grid lines.

Rank 2 matrices encompass most 2D matrices, and I've already covered this case in §3.3.4 and §3.4.1. A rank 1 matrix collapses all of 2D space onto a single line, so for this type of matrix, we should just draw this line.

This code is in `VectorGridPlot.draw_parallel_lines()`. We assemble the matrix $\begin{pmatrix} \texttt{vector\_x} & \texttt{point\_x} \\ \texttt{vector\_y} & \texttt{point\_y} \end{pmatrix}$ (which is actually the matrix used to create the transformation we're trying to render lines for) and use this matrix to check determinant and rank.

```
# 677b38c87bb6722b16aaf35058cf3cef66e43c21
# src/lintrans/gui/plots/classes.py
```

```
177             # If the determinant is 0
178             if abs(vector_x * point_y - vector_y * point_x) < 1e-12:
179                 rank = np.linalg.matrix_rank(
180                     np.array([
181                         [vector_x, point_x],
182                         [vector_y, point_y]
183                     ])
184                 )
185 
186                 # If the matrix is rank 1, then we can draw the column space line
187                 if rank == 1:
188                     self.draw_oblique_line(painter, vector_y / vector_x, 0)
189 
190                 # If the rank is 0, then we don't draw any lines
191                 else:
192                     return
```

Additionally, there was a bug with animating these determinant 0 matrices, since we try to scale the determinant through the animation, as documented in §3.3.6, but when the determinant is 0, this causes issues. To fix this, we just check the `det_target` variable in `LintransMainWindow.animate_expression` and if it's 0, we use the non-scaled version of the matrix.

```
# b889b686d997c2b64124bee786bccba3fc4f6b08
# src/lintrans/gui/main_window.py
```

```
307                 # If we're animating towards a det 0 matrix, then we don't want to scale the
308                 # determinant with the animation, because this makes the process not work
309                 # I'm doing this here rather than wrapping the whole animation logic in an
310                 # if block mainly because this looks nicer than an extra level of indentation
311                 # The extra processing cost is negligible thanks to NumPy's optimizations
312                 if det_target == 0:
313                     matrix_c = matrix_a
314                 else:
315                     matrix_c = scalar * matrix_b
```

3.4.6 Matrices that are too big

One of my friends was playing around with the prototype and she discovered a bug. When trying to render really big matrices, we can get errors like '`OverflowError: argument 3 overflowed: value must be in the range -2147483648 to 2147483647`' because PyQt5 is a wrapper over Qt5, which is a C++ library that uses the C++ `int` type for the `painter.drawLine()` call. This type is a 32-bit integer. Python can store integers of arbitrary precision, but when PyQt5 calls the underlying C++ library code, this gets cast to a C++ `int` and we can get an `OverflowError`.

This isn't a problem with the gridlines, because we only draw them inside the viewport, as discussed in §3.4.1, and these calculations all happen in Python, so integer precision is not a concern. However, when drawing the basis vectors, we just draw them directly, so we'll have to check that they're within the limit.

I'd previously created a `LintransMainWindow.show_error_message()` method for telling the user when they try to take the inverse of a singular matrix[12].

```
# 0f699dd95b6431e95b2311dcb03e7af49c19613f
# src/lintrans/gui/main_window.py

378     def show_error_message(self, title: str, text: str, info: str | None = None) -> None:
379         """Show an error message in a dialog box.
380 
381         :param str title: The window title of the dialog box
382         :param str text: The simple error message
383         :param info: The more informative error message
384         :type info: Optional[str]
385         """
386         dialog = QMessageBox(self)
387         dialog.setIcon(QMessageBox.Critical)
388         dialog.setWindowTitle(title)
389         dialog.setText(text)
390 
391         if info is not None:
392             dialog.setInformativeText(info)
393 
394         dialog.open()
395 
396         dialog.finished.connect(self.update_render_buttons)
```

I then created the `is_matrix_too_big()` method to just check that the elements of the matrix are within the desired bounds. If it returns `True` when we try to render or animate, then we call `show_error_message()`.

```
# 4682a7b225747cfd77aca0fe3abcdd1397b7c5dd
# src/lintrans/gui/main_window.py

407     def is_matrix_too_big(self, matrix: MatrixType) -> bool:
408         """Check if the given matrix will actually fit onto the canvas.
409 
```

[12] This commit didn't get a standalone section in this write-up because it was so small

```
410             Convert the elements of the matrix to canvas coords and make sure they fit within Qt's 32-bit integer limit.
411
412             :param MatrixType matrix: The matrix to check
413             :returns bool: Whether the matrix fits on the canvas
414             """
415             coords: list[tuple[int, int]] = [self.plot.trans_coords(*vector) for vector in matrix.T]
416
417             for x, y in coords:
418                 if not (-2147483648 <= x <= 2147483647 and -2147483648 <= y <= 2147483647):
419                     return True
420
421             return False
```

3.4.7 Creating the `DefineVisuallyDialog`

Next, I wanted to allow the user to define a matrix visually by dragging the basis vectors. To do this, I obviously needed a new `DefineDialog` subclass for it.

```
# 16ca0229aab73b3f4a8fe752dee3608f3ed6ead5
# src/lintrans/gui/dialogs/define_new_matrix.py

135 class DefineVisuallyDialog(DefineDialog):
136     """The dialog class that allows the user to define a matrix visually."""
137
138     def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
139         """Create the widgets and layout of the dialog.
140
141         :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
142         """
143         super().__init__(matrix_wrapper, *args, **kwargs)
144
145         self.setMinimumSize(500, 450)
146
147         # === Create the widgets
148
149         self.combobox_letter.activated.connect(self.show_matrix)
150
151         self.plot = DefineVisuallyWidget(self)
152
153         # === Arrange the widgets
154
155         self.hlay_definition.addWidget(self.plot)
156         self.hlay_definition.setStretchFactor(self.plot, 1)
157
158         self.vlay_all = QVBoxLayout()
159         self.vlay_all.setSpacing(20)
160         self.vlay_all.addLayout(self.hlay_definition)
161         self.vlay_all.addLayout(self.hlay_buttons)
162
163         self.setLayout(self.vlay_all)
164
165         # We load the default matrix A into the plot
166         self.show_matrix(0)
167
168         # We also enable the confirm button, because any visually defined matrix is valid
169         self.button_confirm.setEnabled(True)
170
171     def update_confirm_button(self) -> None:
172         """Enable the confirm button.
173
174         .. note::
175            The confirm button is always enabled in this dialog and this method is never actually used,
176            so it's got an empty body. It's only here because we need to implement the abstract method.
177         """
178
179     def show_matrix(self, index: int) -> None:
180         """Show the selected matrix on the plot. If the matrix is None, show the identity."""
181         matrix = self.matrix_wrapper[ALPHABET_NO_I[index]]
182
```

```
183             if matrix is None:
184                 matrix = self.matrix_wrapper['I']
185
186             self.plot.visualize_matrix_transformation(matrix)
187             self.plot.update()
188
189         def confirm_matrix(self) -> None:
```

This `DefineVisuallyDialog` class just implements the normal methods needed for a `DefineDialog` and has a `plot` attribute to handle drawing graphics and handling mouse movement. After creating the `DefineVisuallyWidget` as a skeleton and doing some more research in the Qt5 docs[19], I renamed the `trans_coords()` methods to `canvas_coords()` to make the intent more clear, and created a `grid_coords()` method.

```
# 417aea6555029b049c470faff18df29f064f6101
# src/lintrans/gui/plots/classes.py

85          def grid_coords(self, x: int, y: int) -> tuple[float, float]:
86              """Convert a coordinate from canvas coords to grid coords.
87
88              :param int x: The x component of the canvas coordinate
89              :param int y: The y component of the canvas coordinate
90              :returns: The resultant grid coordinates
91              :rtype: tuple[float, float]
92              """
93              # We get the maximum grid coords and convert them into canvas coords
94              return (x - self.canvas_origin[0]) / self.grid_spacing, (-y + self.canvas_origin[1]) / self.grid_spacing
```

I then needed to implement the methods to handle mouse movement in the `DefineVisuallyWidget` class. Thankfully, Ross Wilson, the person who helped me learn about the `QWidget.paintEvent()` method in §3.3.1, also wrote an example of draggable points[5]. In my post, I had explained that I needed draggable points on my canvas, and Ross was helpful enough to create an example in their own time. I probably could've worked it out myself eventually, but this example allowed me to learn a lot quicker.

```
# 417aea6555029b049c470faff18df29f064f6101
# src/lintrans/gui/plots/widgets.py

56  class DefineVisuallyWidget(VisualizeTransformationWidget):
57      """This class is the widget that allows the user to visually define a matrix.
58
59      This is just the widget itself. If you want the dialog, use
60      :class:`lintrans.gui.dialogs.define_new_matrix.DefineVisuallyDialog`.
61      """
62
63      def __init__(self, *args, **kwargs):
64          """Create the widget and enable mouse tracking. ``*args`` and ``**kwargs`` are passed to ``super()``."""
65          super().__init__(*args, **kwargs)
66
67          # self.setMouseTracking(True)
68          self.dragged_point: tuple[float, float] | None = None
69
70          # This is the distance that the cursor needs to be from the point to drag it
71          self.epsilon: int = 5
72
73      def mousePressEvent(self, event: QMouseEvent) -> None:
74          """Handle a QMouseEvent when the user pressed a button."""
75          mx = event.x()
76          my = event.y()
77          button = event.button()
78
79          if button != Qt.LeftButton:
80              event.ignore()
81              return
82
83          for point in (self.point_i, self.point_j):
```

```
84              px, py = self.canvas_coords(*point)
85              if abs(px - mx) <= self.epsilon and abs(py - my) <= self.epsilon:
86                  self.dragged_point = point[0], point[1]
87
88          event.accept()
89
90      def mouseReleaseEvent(self, event: QMouseEvent) -> None:
91          """Handle a QMouseEvent when the user release a button."""
92          if event.button() == Qt.LeftButton:
93              self.dragged_point = None
94              event.accept()
95          else:
96              event.ignore()
97
98      def mouseMoveEvent(self, event: QMouseEvent) -> None:
99          """Handle the mouse moving on the canvas."""
100         mx = event.x()
101         my = event.y()
102
103         if self.dragged_point is not None:
104             x, y = self.grid_coords(mx, my)
105
106             if self.dragged_point == self.point_i:
107                 self.point_i = x, y
108
109             elif self.dragged_point == self.point_j:
110                 self.point_j = x, y
111
112             self.dragged_point = x, y
113
114             self.update()
115
116             print(self.dragged_point)
117             print(self.point_i, self.point_j)
118
119             event.accept()
120
121         event.ignore()
```

This snippet has the line '`self.setMouseTracking(True)`' commented out. This line was in the example, but it turns out that I don't want it. Mouse tracking means that a widget will receive a `QMouseEvent` every time the mouse moves. But if it's disabled (the default), then the widget will only receive a `QMouseEvent` for mouse movement when a button is held down at the same time.

I've also left in some print statements on lines 116 and 117. These small oversights are there because I just forgot to remove them before I committed these changes. They were removed 3 commits later.

3.4.8 Fixing a division by zero bug

When drawing the rank line for a determinant 0, rank 1 matrix, we can encounter a division by zero error. I'm sure this originally manifested in a crash with a `ZeroDivisionError` at runtime, but now I can only get a `RuntimeWarning` when running the old code from commit `16ca0229aab73b3f4a8fe752dee3608f3ed6ead5`. Whether it crashes or just warns the user, there is a division by zero bug when trying to render $\begin{pmatrix} k & 0 \\ 0 & 0 \end{pmatrix}$ or $\begin{pmatrix} 0 & 0 \\ 0 & k \end{pmatrix}$. To fix this, I just handled those cases separately in `VectorGridPlot.draw_parallel_lines(⌋)`.

```
# 40bee6461d477a5c767ed132359cd511c0051e3b
# src/lintrans/gui/plots/classes.py

196             # If the matrix is rank 1, then we can draw the column space line
197             if rank == 1:
198                 if abs(vector_x) < 1e-12:
199                     painter.drawLine(self.width() // 2, 0, self.width() // 2, self.height())
```

```
200                 elif abs(vector_y) < 1e-12:
201                     painter.drawLine(0, self.height() // 2, self.width(), self.height() // 2)
202                 else:
203                     self.draw_oblique_line(painter, vector_y / vector_x, 0)
204 
205             # If the rank is 0, then we don't draw any lines
206             else:
207                 return
```

3.4.9 Implementing transitional animation

Currently, all animation animates from $\mathbf{I}$ to the target matrix $\mathbf{T}$. This means it resets the plot at the start. I eventually want an applicative animation system, where the matrix in the box is applied to the current scene. But I also want an option for a transitional animation, where the program animates from the start matrix $\mathbf{S}$ to the target matrix $\mathbf{T}$, and this seems easier to implement, so I'll do it first.

In `LintransMainWindow`, I created a new method called `animate_between_matrices()` and I call it from `animate_expression()`. The maths for smoothening determinants in §3.3.6 assumed the starting matrix had a determinant of 1, but when using transitional animation, this may not always be true.

If we let $\mathbf{S}$ be the starting matrix, and $\mathbf{A}$ be the matrix from the first stage of calculation as specified in §3.3.6, then we want a c such that $\det(c\mathbf{A}) = \det(\mathbf{S})$, so we get $c = \sqrt{\left|\dfrac{\det(\mathbf{S})}{\det(\mathbf{A})}\right|}$ by the identity $\det(c\mathbf{A}) = c^2 \det(\mathbf{A})$.

Following the same logic as in §3.3.6, we can let $\mathbf{B} = c\mathbf{A}$ and then scale it by d to get the same determinant as the target matrix $\mathbf{T}$ and find that $d = \sqrt{\left|\dfrac{\det(\mathbf{T})}{\det(\mathbf{B})}\right|}$. Unlike previously, $\det(\mathbf{B})$ could be any scalar, so we can't simplify our expression for d.

We then scale this with our `proportion` variable p to get a scalar $s = 1 + p\left(\sqrt{\left|\dfrac{\det(\mathbf{T})}{\det(\mathbf{B})}\right|} - 1\right)$ and render $\mathbf{C} = s\mathbf{B}$ on each frame.

In code, that looks like this:

```
# 4017b84fbce67d8e041bc9ce84cefcb0b6e65e1f
# src/lintrans/gui/main_window.py

275     def animate_expression(self) -> None:
276         """Animate from the current matrix to the matrix in the expression box."""
277         self.button_render.setEnabled(False)
278         self.button_animate.setEnabled(False)
279 
280         # Get the target matrix and it's determinant
281         try:
282             matrix_target = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())
283 
284         except linalg.LinAlgError:
285             self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
286             return
287 
288         matrix_start: MatrixType = np.array([
289             [self.plot.point_i[0], self.plot.point_j[0]],
290             [self.plot.point_i[1], self.plot.point_j[1]]
291         ])
292 
293         self.animate_between_matrices(matrix_start, matrix_target)
294 
295         self.button_render.setEnabled(True)
296         self.button_animate.setEnabled(True)
297 
```

```
298         def animate_between_matrices(self, matrix_start: MatrixType, matrix_target: MatrixType, steps: int = 100) ->
            ↪ None:
299             """Animate from the start matrix to the target matrix."""
300             det_target = linalg.det(matrix_target)
301             det_start = linalg.det(matrix_start)
302 
303             for i in range(0, steps + 1):
304                 # This proportion is how far we are through the loop
305                 proportion = i / steps
306 
307                 # matrix_a is the start matrix plus some part of the target, scaled by the proportion
308                 # If we just used matrix_a, then things would animate, but the determinants would be weird
309                 matrix_a = matrix_start + proportion * (matrix_target - matrix_start)
310 
311                 # So to fix the determinant problem, we get the determinant of matrix_a and use it to normalise
312                 det_a = linalg.det(matrix_a)
313 
314                 # For a 2x2 matrix A and a scalar c, we know that det(cA) = c^2 det(A)
315                 # We want B = cA such that det(B) = det(S), where S is the start matrix,
316                 # so then we can scale it with the animation, so we get
317                 # det(cA) = c^2 det(A) = det(S) => c = sqrt(abs(det(S) / det(A)))
318                 # Then we scale A to get the determinant we want, and call that matrix_b
319                 if det_a == 0:
320                     c = 0
321                 else:
322                     c = np.sqrt(abs(det_start / det_a))
323 
324                 matrix_b = c * matrix_a
325                 det_b = linalg.det(matrix_b)
326 
327                 # matrix_c is the final matrix that we then render for this frame
328                 # It's B, but we scale it over time to have the target determinant
329 
330                 # We want some C = dB such that det(C) is some target determinant T
331                 # det(dB) = d^2 det(B) = T => d = sqrt(abs(T / det(B))
332 
333                 # We're also subtracting 1 and multiplying by the proportion and then adding one
334                 # This just scales the determinant along with the animation
335                 scalar = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
336 
337                 # If we're animating towards a det 0 matrix, then we don't want to scale the
338                 # determinant with the animation, because this makes the process not work
339                 # I'm doing this here rather than wrapping the whole animation logic in an
340                 # if block mainly because this looks nicer than an extra level of indentation
341                 # The extra processing cost is negligible thanks to NumPy's optimizations
342                 if det_target == 0:
343                     matrix_c = matrix_a
344                 else:
345                     matrix_c = scalar * matrix_b
346 
347                 if self.is_matrix_too_big(matrix_c):
348                     self.show_error_message('Matrix too big', "This matrix doesn't fit on the canvas")
349                     return
350 
351                 self.plot.visualize_matrix_transformation(matrix_c)
352 
353                 # We schedule the plot to be updated, tell the event loop to
354                 # process events, and asynchronously sleep for 10ms
355                 # This allows for other events to be processed while animating, like zooming in and out
356                 self.plot.update()
```

This change results in an animation system that will transition from the current matrix to whatever the user types into the input box.

3.4.10 Allowing for sequential animation with commas

Applicative animation has two main forms. There's the version where a standard matrix expression gets applied to the current scene, and the kind where the user defines a sequence of matrices and

we animate through the sequence, applying one at a time. Both of these are referenced in success criterion 5.

I want the user to be able to decide if they want applicative animation or transitional animation, so I'll need to create some form of display settings. However, transitional animation doesn't make much sense for sequential animation[13], so I can implement this now.

Applicative animation is just animating from the matrix **C** representing the current scene to the composition **TC** with the target matrix **T**.

We use **TC** instead of **CT** because matrix multiplication can be thought of as applying successive transformations from right to left. **TC** is the same as starting with the identity **I**, applying **C** (to get to the current scene), and then applying **T**.

Doing this in code is very simple. We just split the expression on commas, and then apply each sub-expression to the current scene one by one, pausing on each comma.

```
# 60584d2559cacbf23479a1bebbb986a800a32331
# src/lintrans/gui/main_window.py
```

```
284         def animate_expression(self) -> None:
285             """Animate from the current matrix to the matrix in the expression box."""
286             self.button_render.setEnabled(False)
287             self.button_animate.setEnabled(False)
288 
289             matrix_start: MatrixType = np.array([
290                 [self.plot.point_i[0], self.plot.point_j[0]],
291                 [self.plot.point_i[1], self.plot.point_j[1]]
292             ])
293 
294             text = self.lineedit_expression_box.text()
295 
296             # If there's commas in the expression, then we want to animate each part at a time
297             if ',' in text:
298                 current_matrix = matrix_start
299 
300                 # For each expression in the list, right multiply it by the current matrix,
301                 # and animate from the current matrix to that new matrix
302                 for expr in text.split(',')[::-1]:
303                     new_matrix = self.matrix_wrapper.evaluate_expression(expr) @ current_matrix
304 
305                     self.animate_between_matrices(current_matrix, new_matrix)
306                     current_matrix = new_matrix
307 
308                     # Here we just redraw and allow for other events to be handled while we pause
309                     self.plot.update()
310                     QApplication.processEvents()
311                     QThread.msleep(500)
312 
313             # If there's no commas, then just animate directly from the start to the target
314             else:
315                 # Get the target matrix and it's determinant
316                 try:
317                     matrix_target = self.matrix_wrapper.evaluate_expression(text)
318 
319                 except linalg.LinAlgError:
320                     self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
321                     return
322 
323                 self.animate_between_matrices(matrix_start, matrix_target)
324 
325             self.update_render_buttons()
```

We're deliberately not checking if the sub-expressions are valid here. We would normally validate the expression in `LintransMainWindow.update_render_buttons()` and only allow the user to render or

[13] I have since changed my thoughts on this, and I allowed sequential transitional animation much later, in commit `41907b81661f3878e435b794d9d719491ef14237`

animate an expression if it's valid. Now we have to check all the sub-expressions if the expression contains commas. Additionally, we can only animate these expressions with commas in them, so rendering should be disabled when the expression contains commas.

Compare the old code to the new code:

```
# 4017b84fbce67d8e041bc9ce84cefcb0b6e65e1f
# src/lintrans/gui/main_window.py

    def update_render_buttons(self) -> None:
        """Enable or disable the render and animate buttons according to whether the matrix expression is valid."""
        valid = self.matrix_wrapper.is_valid_expression(self.lineedit_expression_box.text())
        self.button_render.setEnabled(valid)
        self.button_animate.setEnabled(valid)
```

```
# 60584d2559cacbf23479a1bebbb986a800a32331
# src/lintrans/gui/main_window.py

    def update_render_buttons(self) -> None:
        """Enable or disable the render and animate buttons according to whether the matrix expression is valid."""
        text = self.lineedit_expression_box.text()

        if ',' in text:
            self.button_render.setEnabled(False)

            valid = all(self.matrix_wrapper.is_valid_expression(x) for x in text.split(','))
            self.button_animate.setEnabled(valid)

        else:
            valid = self.matrix_wrapper.is_valid_expression(text)
            self.button_render.setEnabled(valid)
            self.button_animate.setEnabled(valid)
```

3.5 Adding display settings

3.5.1 Creating the dataclass

The first step of adding display settings is creating a dataclass to hold all of the settings. This dataclass will hold attributes to manage how a matrix transformation is displayed. Things like whether to show eigenlines or the determinant parallelogram. It will also hold information for animation. We can factor out the code used to smoothen the determinant, as written in §3.3.6, and make it dependant on a `bool` attribute of the `DisplaySettings` dataclass.

This is a standard class rather than some form of singleton to allow different plots to have different display settings. For example, the user might want different settings for the main view and the visual definition dialog. Allowing each instance of a subclass of `VectorGridPlot` to have its own `DisplaySettings` attribute allows for separate settings for separate plots.

However, this class initially just contained attributes relevant to animation, so it was only an attribute on `LintransMainWindow`.

```
# 2041c7a24d963d8d142d6f0f20ec3828ba8257c6
# src/lintrans/gui/settings.py

"""This module contains the :class:`DisplaySettings` class, which holds configuration for display."""

from dataclasses import dataclass


@dataclass
class DisplaySettings:
    """This class simply holds some attributes to configure display."""
```

```
9
10          animate_determinant: bool = True
11          """This controls whether we want the determinant to change smoothly during the animation."""
12
13          applicative_animation: bool = True
14          """There are two types of simple animation, transitional and applicative.
15
16          Let ``C`` be the matrix representing the currently displayed transformation, and let ``T`` be the target matrix.
17          Transitional animation means that we animate directly from ``C`` from ``T``,
18          and applicative animation means that we animate from ``C`` to ``TC``, so we apply ``T`` to ``C``.
19          """
20
21          animation_pause_length: int = 400
22          """This is the number of milliseconds that we wait between animations when using comma syntax."""
```

Once I had the dataclass, I just had to add '`from .settings import DisplaySettings`' to the top of the file, and '`self.display_settings = DisplaySettings()`' to the constructor of `LintransMainWindow`. I could then use the attributes of this dataclass in `animate_expression()`.

```
# 2041c7a24d963d8d142d6f0f20ec3828ba8257c6
# src/lintrans/gui/main_window.py

286         def animate_expression(self) -> None:
287             """Animate from the current matrix to the matrix in the expression box."""
288             self.button_render.setEnabled(False)
289             self.button_animate.setEnabled(False)
290
291             matrix_start: MatrixType = np.array([
292                 [self.plot.point_i[0], self.plot.point_j[0]],
293                 [self.plot.point_i[1], self.plot.point_j[1]]
294             ])
295
296             text = self.lineedit_expression_box.text()
297
298             # If there's commas in the expression, then we want to animate each part at a time
299             if ',' in text:
300                 current_matrix = matrix_start
301
302                 # For each expression in the list, right multiply it by the current matrix,
303                 # and animate from the current matrix to that new matrix
304                 for expr in text.split(',')[::-1]:
305                     new_matrix = self.matrix_wrapper.evaluate_expression(expr) @ current_matrix
306
307                     self.animate_between_matrices(current_matrix, new_matrix)
308                     current_matrix = new_matrix
309
310                     # Here we just redraw and allow for other events to be handled while we pause
311                     self.plot.update()
312                     QApplication.processEvents()
313                     QThread.msleep(self.display_settings.animation_pause_length)
314
315             # If there's no commas, then just animate directly from the start to the target
316             else:
317                 # Get the target matrix and it's determinant
318                 try:
319                     matrix_target = self.matrix_wrapper.evaluate_expression(text)
320
321                 except linalg.LinAlgError:
322                     self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
323                     return
324
325                 # The concept of applicative animation is explained in /gui/settings.py
326                 if self.display_settings.applicative_animation:
327                     matrix_target = matrix_target @ matrix_start
328
329                 self.animate_between_matrices(matrix_start, matrix_target)
330
331             self.update_render_buttons()
```

I also wrapped the main logic of `animate_between_matrices()` in an `if` block to check if the user wants

the determinant to be smoothed.

```python
# 03e154e1326dc256ffc1a539e97d8ef5ec89f6fd
# src/lintrans/gui/main_window.py

    def animate_between_matrices(self, matrix_start: MatrixType, matrix_target: MatrixType, steps: int = 100) ->
    ↪ None:
        """Animate from the start matrix to the target matrix."""
        det_target = linalg.det(matrix_target)
        det_start = linalg.det(matrix_start)

        for i in range(0, steps + 1):
            # This proportion is how far we are through the loop
            proportion = i / steps

            # matrix_a is the start matrix plus some part of the target, scaled by the proportion
            # If we just used matrix_a, then things would animate, but the determinants would be weird
            matrix_a = matrix_start + proportion * (matrix_target - matrix_start)

            if self.display_settings.animate_determinant and det_target != 0:
                # To fix the determinant problem, we get the determinant of matrix_a and use it to normalise
                det_a = linalg.det(matrix_a)

                # For a 2x2 matrix A and a scalar c, we know that det(cA) = c^2 det(A)
                # We want B = cA such that det(B) = det(S), where S is the start matrix,
                # so then we can scale it with the animation, so we get
                # det(cA) = c^2 det(A) = det(S) => c = sqrt(abs(det(S) / det(A)))
                # Then we scale A to get the determinant we want, and call that matrix_b
                if det_a == 0:
                    c = 0
                else:
                    c = np.sqrt(abs(det_start / det_a))

                matrix_b = c * matrix_a
                det_b = linalg.det(matrix_b)

                # matrix_to_render is the final matrix that we then render for this frame
                # It's B, but we scale it over time to have the target determinant

                # We want some C = dB such that det(C) is some target determinant T
                # det(dB) = d^2 det(B) = T => d = sqrt(abs(T / det(B))

                # We're also subtracting 1 and multiplying by the proportion and then adding one
                # This just scales the determinant along with the animation
                scalar = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
                matrix_to_render = scalar * matrix_b

            else:
                matrix_to_render = matrix_a

            if self.is_matrix_too_big(matrix_to_render):
                self.show_error_message('Matrix too big', "This matrix doesn't fit on the canvas")
                return

            self.plot.visualize_matrix_transformation(matrix_to_render)

            # We schedule the plot to be updated, tell the event loop to
            # process events, and asynchronously sleep for 10ms
            # This allows for other events to be processed while animating, like zooming in and out
            self.plot.update()
            QApplication.processEvents()
            QThread.msleep(1000 // steps)
```

References

[1] Alan O'Callaghan (Alanocallaghan). *color-oracle-java*. Version 1.3. URL: https://github.com/Alanocallaghan/color-oracle-java.

[2] D. Dyson (DoctorDalek1963). *lintrans*. URL: https://github.com/DoctorDalek1963/lintrans.

[3] D. Dyson (DoctorDalek1963). *Which framework should I use for creating draggable points and connecting lines on a 2D grid?* 26th Jan. 2022. URL: https://www.reddit.com/r/learnpython/comments/sd2lbr.

[4] Ross Wilson (rzzzwilson). *Python-Etudes/PyQtCustomWidget*. URL: https://gitlab.com/rzzzwilson/python-etudes/-/tree/master/PyQtCustomWidget.

[5] Ross Wilson (rzzzwilson). *Python-Etudes/PyQtCustomWidget - ijvectors.py*. 26th Jan. 2022. URL: https://gitlab.com/rzzzwilson/python-etudes/-/blob/2b43f5d3c95aa4410db5bed77195bf242318a304/PyQtCustomWidget/ijvectors.py.

[6] *2D linear transformation*. URL: https://www.desmos.com/calculator/upooihuy4s.

[7] Grant Sanderson (3blue1brown). *Essence of Linear Algebra*. 6th Aug. 2016. URL: https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab.

[8] H. Hohn et al. *Matrix Vector*. MIT. 2001. URL: https://mathlets.org/mathlets/matrix-vector/.

[9] Jacek Wodecki and ekhumoro. *How to update window in PyQt5?* URL: https://stackoverflow.com/questions/42045676/how-to-update-window-in-pyqt5.

[10] je1324. *Visualizing Linear Transformations*. 15th Mar. 2018. URL: https://www.geogebra.org/m/YCZa8TAH.

[11] Nathaniel Vaughn Kelso and Bernie Jenny. *Color Oracle*. Version 1.3. URL: https://colororacle.org/.

[12] *Normalize a matrix such that the determinat = 1*. ResearchGate. 26th June 2017. URL: https://www.researchgate.net/post/normalize_a_matrix_such_that_the_determinat_1.

[13] *Plotting with Matplotlib. Create PyQt5 plots with the popular Python plotting library*. URL: https://www.pythonguis.com/tutorials/plotting-matplotlib/.

[14] *PyQt5 Graphics View Framework*. The Qt Company. URL: https://doc.qt.io/qtforpython-5/overviews/graphicsview.html.

[15] *Python 3 Data model - special methods*. Python Software Foundation. URL: https://docs.python.org/3/reference/datamodel.html#special-method-names.

[16] *Python 3.10 Downloads*. Python Software Foundation. URL: https://www.python.org/downloads/release/python-3100/.

[17] *Qt5 for Linux/X11*. The Qt Company. URL: https://doc.qt.io/qt-5/linux.html.

[18] *QWheelEvent class*. The Qt Company. URL: https://doc.qt.io/qt-5/qwheelevent.html.

[19] *QWidget Class (`mouseMoveEvent()` method*. The Qt Company. URL: https://doc.qt.io/qt-5/qwidget.html#mouseMoveEvent.

[20] *QWidget Class (`repaint()` method)*. The Qt Company. URL: https://doc.qt.io/qt-5/qwidget.html#repaint.

[21] *QWidget Class (`update()` method)*. The Qt Company. URL: https://doc.qt.io/qt-5/qwidget.html#update.

[22] Shad Sharma. *Linear Transformation Visualizer*. 4th May 2017. URL: https://shad.io/MatVis/.

[23] Rod Stephens. *Draw lines with arrowheads in C#*. 5th Dec. 2014. URL: http://csharphelper.com/howtos/howto_draw_arrows.html.

[24] *The Event System*. The Qt Company. URL: https://doc.qt.io/qt-5/eventsandfilters.html.

[25] *Types of Color Blindness*. National Eye Institute. URL: https://www.nei.nih.gov/learn-about-eye-health/eye-conditions-and-diseases/color-blindness/types-color-blindness.

A Project code

A.1 `__main__.py`

```
#!/usr/bin/env python

# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This module provides a :func:`main` function to interpret command line arguments and run the program."""

from argparse import ArgumentParser
from textwrap import dedent

from lintrans import __version__, gui
from lintrans.crash_reporting import set_excepthook, set_signal_handler


def main() -> None:
    """Interpret program-specific command line arguments and run the main window in most cases.

    If the user supplies ``--help`` or ``--version``, then we simply respond to that and then return.
    If they don't supply either of these, then we run :func:`lintrans.gui.main_window.main`.

    :param List[str] args: The full argument list (including program name)
    """
    parser = ArgumentParser(add_help=False)

    parser.add_argument(
        'filename',
        nargs='?',
        type=str,
        default=None
    )

    parser.add_argument(
        '-h',
        '--help',
        default=False,
        action='store_true'
    )

    parser.add_argument(
        '-V',
        '--version',
        default=False,
        action='store_true'
    )

    parsed_args = parser.parse_args()

    if parsed_args.help:
        print(dedent('''
        Usage: lintrans [option] [filename]

        Arguments:
            filename         The name of a session file to open

        Options:
            -h, --help       Display this help text and exit
            -V, --version    Display the version information and exit'''[1:]))
        return

    if parsed_args.version:
        print(dedent(f'''
        lintrans (version {__version__})
        The linear transformation visualizer

```

```
68              Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
69
70              This program is licensed under GNU GPLv3, available here:
71              <https://www.gnu.org/licenses/gpl-3.0.html>'''[1:]))
72              return
73
74          gui.main(parsed_args.filename)
75
76
77      if __name__ == '__main__':
78          set_excepthook()
79          set_signal_handler()
80          main()
```

A.2 `crash_reporting.py`

```
1       # lintrans - The linear transformation visualizer
2       # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4       # This program is licensed under GNU GPLv3, available here:
5       # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7       """This module provides functions to report crashes and log them.
8
9       The only functions you should be calling directly are :func:`set_excepthook`
10      and :func:`set_signal_handler` to setup handlers for unhandled exceptions
11      and unhandled operating system signals respectively.
12      """
13
14      from __future__ import annotations
15
16      import os
17      import platform
18      import signal
19      import sys
20      from datetime import datetime
21      from signal import SIGABRT, SIGFPE, SIGILL, SIGSEGV, SIGTERM
22      from textwrap import indent
23      from types import FrameType, TracebackType
24      from typing import Type
25
26      from PyQt5.QtCore import PYQT_VERSION_STR, QT_VERSION_STR
27      from PyQt5.QtWidgets import QApplication
28
29      import lintrans
30      from lintrans.typing_ import is_matrix_type
31      from .global_settings import global_settings
32      from .gui.main_window import LintransMainWindow
33
34
35      def _get_datetime_string() -> str:
36          """Get the date and time as a string with a space in the middle."""
37          return datetime.now().strftime('%Y-%m-%d %H:%M:%S')
38
39
40      def _get_main_window() -> LintransMainWindow:
41          """Return the only instance of :class:`~lintrans.gui.main_window.LintransMainWindow`.
42
43          :raises RuntimeError: If there is not exactly 1 instance of
    ↪     :class:`~lintrans.gui.main_window.LintransMainWindow`
44          """
45          widgets = [
46              x for x in QApplication.topLevelWidgets()
47              if isinstance(x, LintransMainWindow)
48          ]
49
50          if len(widgets) != 1:
51              raise RuntimeError(f'Expected 1 widget of type LintransMainWindow but found {len(widgets)}')
52
53          return widgets[0]
```

```
54
55
56      def _get_system_info() -> str:
57          """Return a string of all the system we could gather."""
58          info = 'SYSTEM INFO:\n'
59
60          info += f'  lintrans: {lintrans.__version__}\n'
61          info += f'  Python: {platform.python_version()}\n'
62          info += f'  Qt5: {QT_VERSION_STR}\n'
63          info += f'  PyQt5: {PYQT_VERSION_STR}\n'
64          info += f'  Platform: {platform.platform()}\n'
65
66          info += '\n'
67          return info
68
69
70      def _get_error_origin(
71          *,
72          exc_type: Type[BaseException] | None,
73          exc_value: BaseException | None,
74          traceback: TracebackType | None,
75          signal_number: int | None,
76          stack_frame: FrameType | None
77      ) -> str:
78          """Return a string specifying the full origin of the error, as best as we can determine.
79
80          This function has effectively two signatures. If the fatal error is caused by an exception,
81          then the first 3 arguments will be used to match the signature of :func:`sys.excepthook`.
82          If it's caused by a signal, then the last two will be used to match the signature of the
83          handler in :func:`signal.signal`. This function should never be used outside this file, so
84          we don't account for a mixture of arguments.
85
86          :param exc_type: The type of the exception that caused the crash
87          :param exc_value: The value of the exception itself
88          :param traceback: The traceback object
89          :param signal_number: The number of the signal that caused the crash
90          :param stack_frame: The current stack frame object
91
92          :type exc_type: Type[BaseException] | None
93          :type exc_value: BaseException | None
94          :type traceback: types.TracebackType | None
95          :type signal_number: int | None
96          :type stack_frame: types.FrameType | None
97          """
98          origin = 'CRASH ORIGIN:\n'
99
100         if exc_type is not None and exc_value is not None and traceback is not None:
101             # We want the frame where the exception actually occurred, so we have to descend the traceback
102             # I don't know why we aren't given this traceback in the first place
103             tb = traceback
104             while tb.tb_next is not None:
105                 tb = tb.tb_next
106
107             frame = tb.tb_frame
108
109             origin += f'  Exception "{exc_value}"\n  of type {exc_type.__name__} in call to {frame.f_code.co_name}()\n'
        ↪  \
110                 f'  on line {frame.f_lineno} of {frame.f_code.co_filename}'
111
112         elif signal_number is not None and stack_frame is not None:
113             origin += f'  Signal "{signal.strsignal(signal_number)}" received in call to
            ↪  {stack_frame.f_code.co_name}()\n' \
114                 f'  on line {stack_frame.f_lineno} of {stack_frame.f_code.co_filename}'
115
116         else:
117             origin += '  UNKNOWN (not exception or signal)'
118
119         origin += '\n\n'
120
121         return origin
122
123
124     def _get_display_settings() -> str:
```

```
        """Return a string representing all of the display settings."""
        display_settings = {
            k: v
            for k, v in _get_main_window()._plot.display_settings.__dict__.items()
            if not k.startswith('_')
        }

        string = 'Display settings:\n'

        for setting, value in display_settings.items():
            string += f'  {setting}: {value}\n'

        return string


    def _get_post_mortem() -> str:
        """Return whatever post mortem data we could gather from the window."""
        window = _get_main_window()

        try:
            matrix_wrapper = window._matrix_wrapper
            plot = window._plot
            point_i = plot.point_i
            point_j = plot.point_j

        except (AttributeError, RuntimeError) as e:
            return f'UNABLE TO GET POST MORTEM DATA:\n  {e!r}\n'

        post_mortem = 'Matrix wrapper:\n'

        for matrix_name, matrix_value in matrix_wrapper.get_defined_matrices():
            post_mortem += f'  {matrix_name}: '

            if is_matrix_type(matrix_value):
                post_mortem += f'[{matrix_value[0][0]} {matrix_value[0][1]}; {matrix_value[1][0]} {matrix_value[1][1]}]'
            else:
                post_mortem += f'"{matrix_value}"'

            post_mortem += '\n'

        post_mortem += f'\nExpression box: "{window._lineedit_expression_box.text()}"'
        post_mortem += f'\nCurrently displayed: [{point_i[0]} {point_j[0]}; {point_i[1]} {point_j[1]}]'
        post_mortem += f'\nAnimating (sequence): {window._animating} ({window._animating_sequence})\n'

        post_mortem += f'\nGrid spacing: {plot.grid_spacing}'
        post_mortem += f'\nWindow size: {window.width()} x {window.height()}'
        post_mortem += f'\nViewport size: {plot.width()} x {plot.height()}'
        post_mortem += f'\nGrid corner: {plot._grid_corner()}\n'

        post_mortem += '\n' + _get_display_settings()

        string = 'POST MORTEM:\n'
        string += indent(post_mortem, '  ')
        return string


    def _get_crash_report(datetime_string: str, error_origin: str) -> str:
        """Return a string crash report, ready to be written to a file and stderr.

        :param str datetime_string: The datetime to use in the report; should be the same as the one in the filename
        :param str error_origin: The origin of the error. Get this by calling :func:`_get_error_origin`
        """
        report = f'CRASH REPORT at {datetime_string}\n\n'
        report += _get_system_info()
        report += error_origin
        report += _get_post_mortem()

        return report


    def _report_crash(
        *,
        exc_type: Type[BaseException] | None = None,
```

```
        exc_value: BaseException | None = None,
        traceback: TracebackType | None = None,
        signal_number: int | None = None,
        stack_frame: FrameType | None = None
    ) -> None:
        """Generate a crash report and write it to a log file and stderr.

        See :func:`_get_error_origin` for an explanation of the arguments. Everything is
        handled internally if you just use the public functions :func:`set_excepthook` and
        :func:`set_signal_handler`.
        """
        datetime_string = _get_datetime_string()

        filename = os.path.join(
            global_settings.get_crash_reports_directory(),
            datetime_string.replace(" ", "_") + '.log'
        )
        report = _get_crash_report(
            datetime_string,
            _get_error_origin(
                exc_type=exc_type,
                exc_value=exc_value,
                traceback=traceback,
                signal_number=signal_number,
                stack_frame=stack_frame
            )
        )

        print('\n\n' + report, end='', file=sys.stderr)
        with open(filename, 'w', encoding='utf-8') as f:
            f.write(report)

        sys.exit(255)


    def set_excepthook() -> None:
        """Change :func:`sys.excepthook` to generate a crash report first."""
        def _custom_excepthook(
            exc_type: Type[BaseException],
            exc_value: BaseException,
            traceback: TracebackType | None
        ) -> None:
            _report_crash(exc_type=exc_type, exc_value=exc_value, traceback=traceback)

        sys.excepthook = _custom_excepthook


    def set_signal_handler() -> None:
        """Set the signal handlers to generate crash reports first."""
        def _handler(number, frame) -> None:
            _report_crash(signal_number=number, stack_frame=frame)

        for sig_num in (SIGABRT, SIGFPE, SIGILL, SIGSEGV, SIGTERM):
            if sig_num in signal.valid_signals():
                signal.signal(sig_num, _handler)

        try:
            from signal import SIGQUIT
            signal.signal(SIGQUIT, _handler)
        except ImportError:
            pass
```

A.3 `global_settings.py`

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

```

```python
"""This module provides the :attr:`global_settings` attribute, which should be used to access global settings."""

from __future__ import annotations

import os


class _GlobalSettings:
    """A class to provide global settings that can be shared throughout the app.

    The directory methods are split up into things like :meth:`get_save_directory` and
    :meth:`get_crash_reports_directory` to make sure the directories exist and discourage
    the use of other directories in the root one.

    .. warning::
       This class should never be directly used and should only be
       accessed through the :attr:`global_settings` attribute.
    """

    def __new__(cls) -> _GlobalSettings:
        """Override :meth:`__new__` to implement a singleton. This class will only be created once."""
        # Only create a new instance if we don't already have one
        if not hasattr(cls, '_instance'):
            cls._instance = super(_GlobalSettings, cls).__new__(cls)

        return cls._instance

    def __init__(self) -> None:
        """Create the global settings object and initialize state."""
        # The root directory is OS-dependent
        if os.name == 'posix':
            self._directory = os.path.join(
                os.path.expanduser('~'),
                '.lintrans'
            )

        elif os.name == 'nt':
            self._directory = os.path.join(
                os.path.expandvars('%APPDATA%'),
                'lintrans'
            )

        else:
            # This should be unreachable because the only other option for os.name is 'java'
            # for Jython, but Jython only supports Python 2.7, which has been EOL for a while
            # lintrans is only compatible with Python >= 3.8 anyway
            raise OSError(f'Unrecognised OS "{os.name}"')

        sub_directories = ['saves', 'crash_reports']

        os.makedirs(self._directory, exist_ok=True)
        for sub_directory in sub_directories:
            os.makedirs(os.path.join(self._directory, sub_directory), exist_ok=True)

    def get_save_directory(self) -> str:
        """Return the default directory for save files."""
        return os.path.join(self._directory, 'saves')

    def get_crash_reports_directory(self) -> str:
        """Return the default directory for crash reports."""
        return os.path.join(self._directory, 'crash_reports')


global_settings = _GlobalSettings()
"""This attribute is the only way that global settings should be accessed.

For the private class, see :class:`_GlobalSettings`.
"""
```

A.4 `__init__.py`

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This is the top-level ``lintrans`` package, which contains all the subpackages of the project."""

from . import crash_reporting, global_settings, gui, matrices, typing_

__version__ = '1.0.0-alpha'

__all__ = ['crash_reporting', 'global_settings', 'gui', 'matrices', 'typing_', '__version__']
```

A.5 `gui/validate.py`

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This simple module provides a :class:`MatrixExpressionValidator` class to validate matrix expression input."""

from __future__ import annotations

import re
from typing import Tuple

from PyQt5.QtGui import QValidator

from lintrans.matrices import parse


class MatrixExpressionValidator(QValidator):
    """This class validates matrix expressions in a Qt input box."""

    def validate(self, text: str, pos: int) -> Tuple[QValidator.State, str, int]:
        """Validate the given text according to the rules defined in the :mod:`~lintrans.matrices` module."""
        # We want to extend the naive character class by adding a comma, which isn't
        # normally allowed in expressions, but is allowed for sequential animations
        bad_chars = re.sub(parse.NAIVE_CHARACTER_CLASS[:-1] + ',]', '', text)

        # If there are bad chars, just reject it
        if bad_chars != '':
            return QValidator.Invalid, text, pos

        # Now we need to check if it's actually a valid expression
        if all(parse.validate_matrix_expression(expression) for expression in text.split(',')):
            return QValidator.Acceptable, text, pos

        # Else, if it's got all the right characters but it's not a valid expression
        return QValidator.Intermediate, text, pos
```

A.6 `gui/main_window.py`

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This module provides the :class:`LintransMainWindow` class, which provides the main window for the GUI."""

from __future__ import annotations
```

```
10
11      import os
12      import re
13      import sys
14      import webbrowser
15      from copy import deepcopy
16      from pathlib import Path
17      from pickle import UnpicklingError
18      from typing import List, Optional, Tuple, Type
19
20      import numpy as np
21      from numpy import linalg
22      from numpy.linalg import LinAlgError
23      from PyQt5 import QtWidgets
24      from PyQt5.QtCore import pyqtSlot, QThread
25      from PyQt5.QtGui import QCloseEvent, QIcon, QKeySequence
26      from PyQt5.QtWidgets import (QAction, QApplication, QFileDialog, QHBoxLayout, QMainWindow, QMenu, QMessageBox,
27                                   QPushButton, QShortcut, QSizePolicy, QSpacerItem, QStyleFactory, QVBoxLayout)
28
29      import lintrans
30      from lintrans.global_settings import global_settings
31      from lintrans.matrices import MatrixWrapper
32      from lintrans.matrices.parse import validate_matrix_expression
33      from lintrans.matrices.utility import polar_coords, rotate_coord
34      from lintrans.typing_ import MatrixType, VectorType
35      from .dialogs import (AboutDialog, DefineAsExpressionDialog, DefineMatrixDialog,
36                            DefineNumericallyDialog, DefinePolygonDialog, DefineVisuallyDialog,
37                            DisplaySettingsDialog, FileSelectDialog, InfoPanelDialog)
38      from .plots import MainViewportWidget
39      from .session import Session
40      from .settings import DisplaySettings
41      from .utility import qapp
42      from .validate import MatrixExpressionValidator
43
44
45      class LintransMainWindow(QMainWindow):
46          """This class provides a main window for the GUI using the Qt framework.
47
48          This class should not be used directly, instead call :func:`main` to create the GUI.
49          """
50
51          def __init__(self):
52              """Create the main window object, and create and arrange every widget in it.
53
54              This doesn't show the window, it just constructs it. Use :func:`main` to show the GUI.
55              """
56              super().__init__()
57
58              self._matrix_wrapper = MatrixWrapper()
59
60              self.setWindowTitle('lintrans')
61              self.setMinimumSize(1000, 750)
62
63              path = Path(__file__).parent.absolute() / 'assets' / 'icon.jpg'
64              self.setWindowIcon(QIcon(str(path)))
65
66              self._animating: bool = False
67              self._animating_sequence: bool = False
68              self._reset_during_animation: bool = False
69
70              self._save_filename: Optional[str] = None
71              self._changed_since_save: bool = False
72
73              # === Create menubar
74
75              menubar = QtWidgets.QMenuBar(self)
76
77              menu_file = QMenu(menubar)
78              menu_file.setTitle('&File')
79
80              menu_help = QMenu(menubar)
81              menu_help.setTitle('&Help')
82
```

```
83          action_reset_session = QAction(self)
84          action_reset_session.setText('Reset session')
85          action_reset_session.triggered.connect(self._reset_session)
86
87          action_open = QAction(self)
88          action_open.setText('&Open')
89          action_open.setShortcut('Ctrl+O')
90          action_open.triggered.connect(self._ask_for_session_file)
91
92          action_save = QAction(self)
93          action_save.setText('&Save')
94          action_save.setShortcut('Ctrl+S')
95          action_save.triggered.connect(self._save_session)
96
97          action_save_as = QAction(self)
98          action_save_as.setText('Save as...')
99          action_save_as.setShortcut('Ctrl+Shift+S')
100         action_save_as.triggered.connect(self._save_session_as)
101
102         action_quit = QAction(self)
103         action_quit.setText('&Quit')
104         action_quit.triggered.connect(self.close)
105
106         # If this is an old release, use the docs for this release. Else, use the latest docs
107         # We use the latest because most use cases for non-stable releases will be in development and testing
108         docs_link = 'https://lintrans.readthedocs.io/en/'
109
110         if re.match(r'^\d+\.\d+\.\d+$', lintrans.__version__):
111             docs_link += 'v' + lintrans.__version__
112         else:
113             docs_link += 'latest'
114
115         action_tutorial = QAction(self)
116         action_tutorial.setText('&Tutorial')
117         action_tutorial.setShortcut('F1')
118         action_tutorial.triggered.connect(
119             lambda: webbrowser.open_new_tab(docs_link + '/tutorial/index.html')
120         )
121
122         action_docs = QAction(self)
123         action_docs.setText('&Docs')
124         action_docs.triggered.connect(
125             lambda: webbrowser.open_new_tab(docs_link + '/backend/lintrans.html')
126         )
127
128         menu_feedback = QMenu(menu_help)
129         menu_feedback.setTitle('Give feedback')
130
131         action_bug_report = QAction(self)
132         action_bug_report.setText('Report a bug')
133         action_bug_report.triggered.connect(
134             lambda: webbrowser.open_new_tab('https://forms.gle/Q82cLTtgPLcV4xQD6')
135         )
136
137         action_suggest_feature = QAction(self)
138         action_suggest_feature.setText('Suggest a new feature')
139         action_suggest_feature.triggered.connect(
140             lambda: webbrowser.open_new_tab('https://forms.gle/mVWbHiMBw9Zq5Ze37')
141         )
142
143         menu_feedback.addAction(action_bug_report)
144         menu_feedback.addAction(action_suggest_feature)
145
146         action_about = QAction(self)
147         action_about.setText('&About')
148         action_about.triggered.connect(lambda: AboutDialog(self).open())
149
150         menu_file.addAction(action_reset_session)
151         menu_file.addAction(action_open)
152         menu_file.addSeparator()
153         menu_file.addAction(action_save)
154         menu_file.addAction(action_save_as)
155         menu_file.addSeparator()
```

```
156             menu_file.addAction(action_quit)
157
158             menu_help.addAction(action_tutorial)
159             menu_help.addAction(action_docs)
160             menu_help.addSeparator()
161             menu_help.addMenu(menu_feedback)
162             menu_help.addSeparator()
163             menu_help.addAction(action_about)
164
165             menubar.addAction(menu_file.menuAction())
166             menubar.addAction(menu_help.menuAction())
167
168             self.setMenuBar(menubar)
169
170             # === Create widgets
171
172             # Left layout: the plot and input box
173
174             self._plot = MainViewportWidget(self, display_settings=DisplaySettings(), polygon_points=[])
175
176             self._lineedit_expression_box = QtWidgets.QLineEdit(self)
177             self._lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
178             self._lineedit_expression_box.setValidator(MatrixExpressionValidator(self))
179             self._lineedit_expression_box.textChanged.connect(self._update_render_buttons)
180
181             # Right layout: all the buttons
182
183             # Misc buttons
184
185             button_define_polygon = QPushButton(self)
186             button_define_polygon.setText('Define polygon')
187             button_define_polygon.clicked.connect(self._dialog_define_polygon)
188             button_define_polygon.setToolTip('Define a polygon to view its transformation<br><b>(Ctrl + P)</b>')
189             QShortcut(QKeySequence('Ctrl+P'), self).activated.connect(button_define_polygon.click)
190
191             self._button_change_display_settings = QPushButton(self)
192             self._button_change_display_settings.setText('Change\ndisplay settings')
193             self._button_change_display_settings.clicked.connect(self._dialog_change_display_settings)
194             self._button_change_display_settings.setToolTip(
195                 "Change which things are rendered and how they're rendered<br><b>(Ctrl + D)</b>"
196             )
197             QShortcut(QKeySequence('Ctrl+D'), self).activated.connect(self._button_change_display_settings.click)
198
199             button_reset_zoom = QPushButton(self)
200             button_reset_zoom.setText('Reset zoom')
201             button_reset_zoom.clicked.connect(self._reset_zoom)
202             button_reset_zoom.setToolTip('Reset the zoom level back to normal<br><b>(Ctrl + Shift + R)</b>')
203             QShortcut(QKeySequence('Ctrl+Shift+R'), self).activated.connect(button_reset_zoom.click)
204
205             # Define new matrix buttons and their groupbox
206
207             self._button_define_visually = QPushButton(self)
208             self._button_define_visually.setText('Visually')
209             self._button_define_visually.setToolTip('Drag the basis vectors<br><b>(Alt + 1)</b>')
210             self._button_define_visually.clicked.connect(lambda: self._dialog_define_matrix(DefineVisuallyDialog))
211             QShortcut(QKeySequence('Alt+1'), self).activated.connect(self._button_define_visually.click)
212
213             self._button_define_numerically = QPushButton(self)
214             self._button_define_numerically.setText('Numerically')
215             self._button_define_numerically.setToolTip('Define a matrix just with numbers<br><b>(Alt + 2)</b>')
216             self._button_define_numerically.clicked.connect(lambda: self._dialog_define_matrix(DefineNumericallyDialog))
217             QShortcut(QKeySequence('Alt+2'), self).activated.connect(self._button_define_numerically.click)
218
219             self._button_define_as_expression = QPushButton(self)
220             self._button_define_as_expression.setText('As an expression')
221             self._button_define_as_expression.setToolTip('Define a matrix in terms of other matrices<br><b>(Alt +
                ↪  3)</b>')
222             self._button_define_as_expression.clicked.connect(
223                 lambda: self._dialog_define_matrix(DefineAsExpressionDialog)
224             )
225             QShortcut(QKeySequence('Alt+3'), self).activated.connect(self._button_define_as_expression.click)
226
227             vlay_define_new_matrix = QVBoxLayout()
```

```
228             vlay_define_new_matrix.setSpacing(20)
229             vlay_define_new_matrix.addWidget(self._button_define_visually)
230             vlay_define_new_matrix.addWidget(self._button_define_numerically)
231             vlay_define_new_matrix.addWidget(self._button_define_as_expression)
232 
233             groupbox_define_new_matrix = QtWidgets.QGroupBox('Define a new matrix', self)
234             groupbox_define_new_matrix.setLayout(vlay_define_new_matrix)
235 
236             # Info panel button
237 
238             self._button_info_panel = QPushButton(self)
239             self._button_info_panel.setText('Show defined matrices')
240             self._button_info_panel.clicked.connect(
241                 # We have to use a lambda instead of 'InfoPanelDialog(self.matrix_wrapper, self).open' here
242                 # because that would create an unnamed instance of InfoPanelDialog when LintransMainWindow is
243                 # constructed, but we need to create a new instance every time to keep self.matrix_wrapper up to date
244                 lambda: InfoPanelDialog(self._matrix_wrapper, self).open()
245             )
246             self._button_info_panel.setToolTip(
247                 'Open an info panel with all matrices that have been defined in this session<br><b>(Ctrl + M)</b>'
248             )
249             QShortcut(QKeySequence('Ctrl+M'), self).activated.connect(self._button_info_panel.click)
250 
251             # Render buttons
252 
253             button_reset = QPushButton(self)
254             button_reset.setText('Reset')
255             button_reset.clicked.connect(self._reset_transformation)
256             button_reset.setToolTip('Reset the visualized transformation back to the identity<br><b>(Ctrl + R)</b>')
257             QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(button_reset.click)
258 
259             self._button_render = QPushButton(self)
260             self._button_render.setText('Render')
261             self._button_render.setEnabled(False)
262             self._button_render.clicked.connect(self._render_expression)
263             self._button_render.setToolTip('Render the expression<br><b>(Ctrl + Enter)</b>')
264             QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self._button_render.click)
265 
266             self._button_animate = QPushButton(self)
267             self._button_animate.setText('Animate')
268             self._button_animate.setEnabled(False)
269             self._button_animate.clicked.connect(self._animate_expression)
270             self._button_animate.setToolTip('Animate the expression<br><b>(Ctrl + Shift + Enter)</b>')
271             QShortcut(QKeySequence('Ctrl+Shift+Return'), self).activated.connect(self._button_animate.click)
272 
273             # === Arrange widgets
274 
275             vlay_left = QVBoxLayout()
276             vlay_left.addWidget(self._plot)
277             vlay_left.addWidget(self._lineedit_expression_box)
278 
279             vlay_misc_buttons = QVBoxLayout()
280             vlay_misc_buttons.setSpacing(20)
281             vlay_misc_buttons.addWidget(button_define_polygon)
282             vlay_misc_buttons.addWidget(self._button_change_display_settings)
283             vlay_misc_buttons.addWidget(button_reset_zoom)
284 
285             vlay_info_buttons = QVBoxLayout()
286             vlay_info_buttons.setSpacing(20)
287             vlay_info_buttons.addWidget(self._button_info_panel)
288 
289             vlay_render = QVBoxLayout()
290             vlay_render.setSpacing(20)
291             vlay_render.addWidget(button_reset)
292             vlay_render.addWidget(self._button_animate)
293             vlay_render.addWidget(self._button_render)
294 
295             vlay_right = QVBoxLayout()
296             vlay_right.setSpacing(50)
297             vlay_right.addLayout(vlay_misc_buttons)
298             vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
299             vlay_right.addWidget(groupbox_define_new_matrix)
300             vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
```

```
            vlay_right.addLayout(vlay_info_buttons)
            vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
            vlay_right.addLayout(vlay_render)

            hlay_all = QHBoxLayout()
            hlay_all.setSpacing(15)
            hlay_all.addLayout(vlay_left)
            hlay_all.addLayout(vlay_right)

            central_widget = QtWidgets.QWidget()
            central_widget.setLayout(hlay_all)
            central_widget.setContentsMargins(10, 10, 10, 10)

            self.setCentralWidget(central_widget)

        def closeEvent(self, event: QCloseEvent) -> None:
            """Handle a :class:`QCloseEvent` by confirming if the user wants to save, and cancelling animation."""
            if self._save_filename is None or not self._changed_since_save:
                self._animating = False
                self._animating_sequence = False
                event.accept()
                return

            dialog = QMessageBox(self)
            dialog.setIcon(QMessageBox.Question)
            dialog.setWindowTitle('Save changes?')
            dialog.setText(f"If you don't save, then changes made to {self._save_filename} will be lost.")
            dialog.setStandardButtons(QMessageBox.Save | QMessageBox.Discard | QMessageBox.Cancel)
            dialog.setDefaultButton(QMessageBox.Save)

            pressed_button = dialog.exec()

            if pressed_button == QMessageBox.Save:
                self._save_session()

            if pressed_button in (QMessageBox.Save, QMessageBox.Discard):
                self._animating = False
                self._animating_sequence = False
                event.accept()
            else:
                event.ignore()

        def _update_render_buttons(self) -> None:
            """Enable or disable the render and animate buttons according to whether the matrix expression is valid."""
            text = self._lineedit_expression_box.text()

            # Let's say that the user defines a non-singular matrix A, then defines B as A^-1
            # If they then redefine A and make it singular, then we get a LinAlgError when
            # trying to evaluate an expression with B in it
            # To fix this, we just do naive validation rather than aware validation
            if ',' in text:
                self._button_render.setEnabled(False)

                try:
                    valid = all(self._matrix_wrapper.is_valid_expression(x) for x in text.split(','))
                except LinAlgError:
                    valid = all(validate_matrix_expression(x) for x in text.split(','))

                self._button_animate.setEnabled(valid)

            else:
                try:
                    valid = self._matrix_wrapper.is_valid_expression(text)
                except LinAlgError:
                    valid = validate_matrix_expression(text)

                self._button_render.setEnabled(valid)
                self._button_animate.setEnabled(valid)

        @pyqtSlot()
        def _reset_zoom(self) -> None:
            """Reset the zoom level back to normal."""
            self._plot.grid_spacing = self._plot.DEFAULT_GRID_SPACING
```

```
374             self._plot.update()
375
376         @pyqtSlot()
377         def _reset_transformation(self) -> None:
378             """Reset the visualized transformation back to the identity."""
379             if self._animating or self._animating_sequence:
380                 self._reset_during_animation = True
381
382             self._animating = False
383             self._animating_sequence = False
384
385             self._plot.plot_matrix(self._matrix_wrapper['I'])
386             self._plot.update()
387
388         @pyqtSlot()
389         def _render_expression(self) -> None:
390             """Render the transformation given by the expression in the input box."""
391             try:
392                 matrix = self._matrix_wrapper.evaluate_expression(self._lineedit_expression_box.text())
393
394             except LinAlgError:
395                 self._show_error_message('Singular matrix', 'Cannot take inverse of singular matrix.')
396                 return
397
398             if self._is_matrix_too_big(matrix):
399                 self._show_error_message('Matrix too big', "This matrix doesn't fit on the canvas.")
400                 return
401
402             self._plot.plot_matrix(matrix)
403             self._plot.update()
404
405         @pyqtSlot()
406         def _animate_expression(self) -> None:
407             """Animate from the current matrix to the matrix in the expression box."""
408             self._button_render.setEnabled(False)
409             self._button_animate.setEnabled(False)
410
411             matrix_start: MatrixType = np.array([
412                 [self._plot.point_i[0], self._plot.point_j[0]],
413                 [self._plot.point_i[1], self._plot.point_j[1]]
414             ])
415
416             text = self._lineedit_expression_box.text()
417
418             # If there's commas in the expression, then we want to animate each part at a time
419             if ',' in text:
420                 current_matrix = matrix_start
421                 self._animating_sequence = True
422
423                 # For each expression in the list, right multiply it by the current matrix,
424                 # and animate from the current matrix to that new matrix
425                 for expr in text.split(',')[::-1]:
426                     if not self._animating_sequence:
427                         break
428
429                     try:
430                         new_matrix = self._matrix_wrapper.evaluate_expression(expr)
431
432                         if self._plot.display_settings.applicative_animation:
433                             new_matrix = new_matrix @ current_matrix
434                     except LinAlgError:
435                         self._show_error_message('Singular matrix', 'Cannot take inverse of singular matrix.')
436                         return
437
438                     self._animate_between_matrices(current_matrix, new_matrix)
439                     current_matrix = new_matrix
440
441                     # Here we just redraw and allow for other events to be handled while we pause
442                     self._plot.update()
443                     QApplication.processEvents()
444                     QThread.msleep(self._plot.display_settings.animation_pause_length)
445
446                 self._animating_sequence = False
```

```
447
448             # If there's no commas, then just animate directly from the start to the target
449             else:
450                 # Get the target matrix and its determinant
451                 try:
452                     matrix_target = self._matrix_wrapper.evaluate_expression(text)
453
454                 except LinAlgError:
455                     self._show_error_message('Singular matrix', 'Cannot take inverse of singular matrix.')
456                     return
457
458                 # The concept of applicative animation is explained in /gui/settings.py
459                 if self._plot.display_settings.applicative_animation:
460                     matrix_target = matrix_target @ matrix_start
461
462                 # If we want a transitional animation and we're animating the same matrix, then restart the animation
463                 # We use this check rather than equality because of small floating point errors
464                 elif (abs(matrix_start - matrix_target) < 1e-12).all():
465                     matrix_start = self._matrix_wrapper['I']
466
467                     # We pause here for 200 ms to make the animation look a bit nicer
468                     self._plot.plot_matrix(matrix_start)
469                     self._plot.update()
470                     QApplication.processEvents()
471                     QThread.msleep(200)
472
473                 self._animate_between_matrices(matrix_start, matrix_target)
474
475             self._update_render_buttons()
476
477         def _get_animation_frame(self, start: MatrixType, target: MatrixType, proportion: float) -> MatrixType:
478             """Get the matrix to render for this frame of the animation.
479
480             This method will smoothen the determinant if that setting in enabled and if the determinant is positive.
481             It also animates rotation-like matrices using a logarithmic spiral to rotate around and scale continuously.
482             Essentially, it just makes things look good when animating.
483
484             :param MatrixType start: The starting matrix
485             :param MatrixType start: The target matrix
486             :param float proportion: How far we are through the loop
487             """
488             det_target = linalg.det(target)
489             det_start = linalg.det(start)
490
491             # This is the matrix that we're applying to get from start to target
492             # We want to check if it's rotation-like
493             if linalg.det(start) == 0:
494                 matrix_application = None
495             else:
496                 matrix_application = target @ linalg.inv(start)
497
498             # For a matrix to represent a rotation, it must have a positive determinant,
499             # its vectors must be perpendicular, the same length, and at right angles
500             # The checks for 'abs(value) < 1e-10' are to account for floating point error
501             if matrix_application is not None \
502                     and self._plot.display_settings.smoothen_determinant \
503                     and linalg.det(matrix_application) > 0 \
504                     and abs(np.dot(matrix_application.T[0], matrix_application.T[1])) < 1e-10 \
505                     and abs(np.hypot(*matrix_application.T[0]) - np.hypot(*matrix_application.T[1])) < 1e-10:
506                 rotation_vector: VectorType = matrix_application.T[0]  # Take the i column
507                 radius, angle = polar_coords(*rotation_vector)
508
509                 # We want the angle to be in [-pi, pi), so we have to subtract 2pi from it if it's too big
510                 if angle > np.pi:
511                     angle -= 2 * np.pi
512
513                 i: VectorType = start.T[0]
514                 j: VectorType = start.T[1]
515
516                 # Scale the coords with a list comprehension
517                 # It's a bit janky, but rotate_coords() will always return a 2-tuple,
518                 # so new_i and new_j will always be lists of length 2
519                 scale = (radius - 1) * proportion + 1
```

```
            new_i = [scale * c for c in rotate_coord(i[0], i[1], angle * proportion)]
            new_j = [scale * c for c in rotate_coord(j[0], j[1], angle * proportion)]

            return np.array(
                [
                    [new_i[0], new_j[0]],
                    [new_i[1], new_j[1]]
                ]
            )

        # matrix_a is the start matrix plus some part of the target, scaled by the proportion
        # If we just used matrix_a, then things would animate, but the determinants would be weird
        matrix_a = start + proportion * (target - start)

        if not self._plot.display_settings.smoothen_determinant or det_start * det_target <= 0:
            return matrix_a

        # To fix the determinant problem, we get the determinant of matrix_a and use it to normalize
        det_a = linalg.det(matrix_a)

        # For a 2x2 matrix A and a scalar c, we know that det(cA) = c^2 det(A)
        # We want B = cA such that det(B) = det(S), where S is the start matrix,
        # so then we can scale it with the animation, so we get
        # det(cA) = c^2 det(A) = det(S) => c = sqrt(abs(det(S) / det(A)))
        # Then we scale A to get the determinant we want, and call that matrix_b
        if det_a == 0:
            c = 0
        else:
            c = np.sqrt(abs(det_start / det_a))

        matrix_b = c * matrix_a
        det_b = linalg.det(matrix_b)

        # We want to return B, but we have to scale it over time to have the target determinant

        # We want some C = dB such that det(C) is some target determinant T
        # det(dB) = d^2 det(B) = T => d = sqrt(abs(T / det(B))

        # We're also subtracting 1 and multiplying by the proportion and then adding one
        # This just scales the determinant along with the animation

        # That is all of course, if we can do that
        # We'll crash if we try to do this with det(B) == 0
        if det_b == 0:
            return matrix_a

        scalar = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
        return scalar * matrix_b

    def _animate_between_matrices(self, matrix_start: MatrixType, matrix_target: MatrixType) -> None:
        """Animate from the start matrix to the target matrix."""
        self._animating = True

        # Making steps depend on animation_time ensures a smooth animation without
        # massive overheads for small animation times
        steps = self._plot.display_settings.animation_time // 10

        for i in range(0, steps + 1):
            if not self._animating:
                break

            matrix_to_render = self._get_animation_frame(matrix_start, matrix_target, i / steps)

            if self._is_matrix_too_big(matrix_to_render):
                self._show_error_message('Matrix too big', "This matrix doesn't fit on the canvas.")
                self._animating = False
                self._animating_sequence = False
                return

            self._plot.plot_matrix(matrix_to_render)

            # We schedule the plot to be updated, tell the event loop to
            # process events, and asynchronously sleep for 10ms
```

```
593                 # This allows for other events to be processed while animating, like zooming in and out
594                 self._plot.update()
595                 QApplication.processEvents()
596                 QThread.msleep(self._plot.display_settings.animation_time // steps)
597 
598             if not self._reset_during_animation:
599                 self._plot.plot_matrix(matrix_target)
600             else:
601                 self._plot.plot_matrix(self._matrix_wrapper['I'])
602 
603             self._plot.update()
604 
605             self._animating = False
606             self._reset_during_animation = False
607 
608         @pyqtSlot(DefineMatrixDialog)
609         def _dialog_define_matrix(self, dialog_class: Type[DefineMatrixDialog]) -> None:
610             """Open a generic definition dialog to define a new matrix.
611 
612             The class for the desired dialog is passed as an argument. We create an
613             instance of this class and the dialog is opened asynchronously and modally
614             (meaning it blocks interaction with the main window) with the proper method
615             connected to the :meth:`QDialog.accepted` signal.
616 
617             .. note:: ``dialog_class`` must subclass
    ↪ :class:`~lintrans.gui.dialogs.define_new_matrix.DefineMatrixDialog`.
618 
619             :param dialog_class: The dialog class to instantiate
620             :type dialog_class: Type[lintrans.gui.dialogs.define_new_matrix.DefineMatrixDialog]
621             """
622             # We create a dialog with a deepcopy of the current matrix_wrapper
623             # This avoids the dialog mutating this one
624             dialog: DefineMatrixDialog
625 
626             if dialog_class == DefineVisuallyDialog:
627                 dialog = DefineVisuallyDialog(
628                     self,
629                     matrix_wrapper=deepcopy(self._matrix_wrapper),
630                     display_settings=self._plot.display_settings,
631                     polygon_points=self._plot.polygon_points
632                 )
633             else:
634                 dialog = dialog_class(self, matrix_wrapper=deepcopy(self._matrix_wrapper))
635 
636             # .open() is asynchronous and doesn't spawn a new event loop, but the dialog is still modal (blocking)
637             dialog.open()
638 
639             # So we have to use the accepted signal to call a method when the user accepts the dialog
640             dialog.accepted.connect(self._assign_matrix_wrapper)
641 
642         @pyqtSlot()
643         def _assign_matrix_wrapper(self) -> None:
644             """Assign a new value to ``self._matrix_wrapper`` and give the expression box focus."""
645             self._matrix_wrapper = self.sender().matrix_wrapper
646             self._lineedit_expression_box.setFocus()
647             self._update_render_buttons()
648 
649             self._changed_since_save = True
650             self._update_window_title()
651 
652         @pyqtSlot()
653         def _dialog_change_display_settings(self) -> None:
654             """Open the dialog to change the display settings."""
655             dialog = DisplaySettingsDialog(self, display_settings=self._plot.display_settings)
656             dialog.open()
657             dialog.accepted.connect(self._assign_display_settings)
658 
659         @pyqtSlot()
660         def _assign_display_settings(self) -> None:
661             """Assign a new value to ``self._plot.display_settings`` and give the expression box focus."""
662             self._plot.display_settings = self.sender().display_settings
663             self._plot.update()
664             self._lineedit_expression_box.setFocus()
```

```
665             self._update_render_buttons()
666
667         @pyqtSlot()
668         def _dialog_define_polygon(self) -> None:
669             """Open the dialog to define a polygon."""
670             dialog = DefinePolygonDialog(self, polygon_points=self._plot.polygon_points)
671             dialog.open()
672             dialog.accepted.connect(self._assign_polygon_points)
673
674         @pyqtSlot()
675         def _assign_polygon_points(self) -> None:
676             """Assign a new value to ``self._plot.polygon_points`` and give the expression box focus."""
677             self._plot.polygon_points = self.sender().polygon_points
678             self._plot.update()
679             self._lineedit_expression_box.setFocus()
680             self._update_render_buttons()
681
682             self._changed_since_save = True
683             self._update_window_title()
684
685         def _show_error_message(self, title: str, text: str, info: str | None = None, *, warning: bool = False) -> None:
686             """Show an error message in a dialog box.
687
688             :param str title: The window title of the dialog box
689             :param str text: The simple error message
690             :param info: The more informative error message
691             :type info: Optional[str]
692             """
693             dialog = QMessageBox(self)
694             dialog.setWindowTitle(title)
695             dialog.setText(text)
696
697             if warning:
698                 dialog.setIcon(QMessageBox.Warning)
699             else:
700                 dialog.setIcon(QMessageBox.Critical)
701
702             if info is not None:
703                 dialog.setInformativeText(info)
704
705             dialog.open()
706
707             # This is `finished` rather than `accepted` because we want to update the buttons no matter what
708             dialog.finished.connect(self._update_render_buttons)
709
710         def _is_matrix_too_big(self, matrix: MatrixType) -> bool:
711             """Check if the given matrix will actually fit onto the canvas.
712
713             Convert the elements of the matrix to canvas coords and make sure they fit within Qt's 32-bit integer limit.
714
715             :param MatrixType matrix: The matrix to check
716             :returns bool: Whether the matrix is too big to fit on the canvas
717             """
718             coords: List[Tuple[int, int]] = [self._plot.canvas_coords(*vector) for vector in matrix.T]
719
720             for x, y in coords:
721                 if not (-2147483648 <= x <= 2147483647 and -2147483648 <= y <= 2147483647):
722                     return True
723
724             return False
725
726         def _update_window_title(self) -> None:
727             """Update the window title to reflect whether the session has changed since it was last saved."""
728             title = 'lintrans'
729
730             if self._save_filename:
731                 title = os.path.split(self._save_filename)[-1] + ' - ' + title
732
733                 if self._changed_since_save:
734                     title = '*' + title
735
736             self.setWindowTitle(title)
737
```

```
    def _reset_session(self) -> None:
        """Ask the user if they want to reset the current session.

        Resetting the session means setting the matrix wrapper to a new instance, and rendering I.
        """
        dialog = QMessageBox(self)
        dialog.setIcon(QMessageBox.Question)
        dialog.setWindowTitle('Reset the session?')
        dialog.setText('Are you sure you want to reset the current session?')
        dialog.setStandardButtons(QMessageBox.Yes | QMessageBox.No)
        dialog.setDefaultButton(QMessageBox.No)

        if dialog.exec() == QMessageBox.Yes:
            self._matrix_wrapper = MatrixWrapper()

            self._lineedit_expression_box.setText('I')
            self._render_expression()
            self._lineedit_expression_box.setText('')
            self._lineedit_expression_box.setFocus()
            self._update_render_buttons()

            self._save_filename = None
            self._changed_since_save = False
            self._update_window_title()

    def open_session_file(self, filename: str) -> None:
        """Open the given session file.

        If the selected file is not a valid lintrans session file, we just show an error message,
        but if it's valid, we load it and set it as the default filename for saving.
        """
        try:
            session, version, extra_attrs = Session.load_from_file(filename)

        # load_from_file() can raise errors if the contents is not a valid pickled Python object,
        # or if the pickled Python object is of the wrong type
        except (AttributeError, EOFError, FileNotFoundError, ValueError, UnpicklingError):
            self._show_error_message(
                'Invalid file contents',
                'This is not a valid lintrans session file.',
                'Not all .lt files are lintrans session files. This file was probably created by an unrelated '
                'program.'
            )
            return

        missing_parts = False

        if session.matrix_wrapper is not None:
            self._matrix_wrapper = session.matrix_wrapper
        else:
            missing_parts = True  # type: ignore[unreachable]

        if session.polygon_points is not None:
            self._plot.polygon_points = session.polygon_points
        else:
            missing_parts = True  # type: ignore[unreachable]

        if missing_parts:
            if version != lintrans.__version__:
                info = f"This may be a version conflict. This file was saved with lintrans v{version} " \
                       f"but you're running lintrans v{lintrans.__version__}."
            else:
                info = None

            self._show_error_message(
                'Session file missing parts',
                'This session file is missing certain elements. It may not work correctly.',
                info,
                warning=True
            )
        elif extra_attrs:
            if version != lintrans.__version__:
                info = f"This may be a version conflict. This file was saved with lintrans v{version} " \
```

```
                        f"but you're running lintrans v{lintrans.__version__}."
                else:
                    info = None

                self._show_error_message(
                    'Session file has extra parts',
                    'This session file has more parts than expected. It will work correctly, '
                    'but you might be missing some features.',
                    info,
                    warning=True
                )

            self._lineedit_expression_box.setText('I')
            self._render_expression()
            self._lineedit_expression_box.setText('')
            self._lineedit_expression_box.setFocus()
            self._update_render_buttons()

            # Set this as the default filename if we could read it properly
            self._save_filename = filename
            self._changed_since_save = False
            self._update_window_title()

        @pyqtSlot()
        def _ask_for_session_file(self) -> None:
            """Ask the user to select a session file, and then open it and load the session."""
            dialog = QFileDialog(
                self,
                'Open a session',
                global_settings.get_save_directory(),
                'lintrans sessions (*.lt)'
            )
            dialog.setAcceptMode(QFileDialog.AcceptOpen)
            dialog.setFileMode(QFileDialog.ExistingFile)
            dialog.setViewMode(QFileDialog.List)

            if dialog.exec():
                self.open_session_file(dialog.selectedFiles()[0])

        @pyqtSlot()
        def _save_session(self) -> None:
            """Save the session to the given file.

            If ``self._save_filename`` is ``None``, then call :meth:`_save_session_as` and return.
            """
            if self._save_filename is None:
                self._save_session_as()
                return

            Session(
                matrix_wrapper=self._matrix_wrapper,
                polygon_points=self._plot.polygon_points
            ).save_to_file(self._save_filename)

            self._changed_since_save = False
            self._update_window_title()

        @pyqtSlot()
        def _save_session_as(self) -> None:
            """Ask the user for a file to save the session to, and then call :meth:`_save_session`.

            .. note::
               If the user doesn't select a file to save the session to, then the session
               just doesn't get saved, and :meth:`_save_session` is never called.
            """
            dialog = FileSelectDialog(
                self,
                'Save this session',
                global_settings.get_save_directory(),
                'lintrans sessions (*.lt)'
            )
            dialog.setAcceptMode(QFileDialog.AcceptSave)
            dialog.setFileMode(QFileDialog.AnyFile)
```

```
884             dialog.setViewMode(QFileDialog.List)
885             dialog.setDefaultSuffix('.lt')
886
887             if dialog.exec():
888                 filename = dialog.selectedFiles()[0]
889                 self._save_filename = filename
890                 self._save_session()
891
892
893     def main(filename: Optional[str]) -> None:
894         """Run the GUI by creating and showing an instance of :class:`LintransMainWindow`.
895
896         :param Optional[str] filename: A session file to optionally open at startup
897         """
898         app = QApplication([])
899         app.setApplicationName('lintrans')
900         app.setApplicationVersion(lintrans.__version__)
901
902         qapp().setStyle(QStyleFactory.create('fusion'))
903
904         window = LintransMainWindow()
905         window.show()
906
907         if filename:
908             window.open_session_file(filename)
909
910         sys.exit(app.exec_())
```

A.7 gui/session.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides the :class:`Session` class, which provides a way to save and load sessions."""
8
9   from __future__ import annotations
10
11  import os
12  import pathlib
13  import pickle
14  from collections import defaultdict
15  from typing import Any, DefaultDict, List, Tuple
16
17  import lintrans
18  from lintrans.matrices import MatrixWrapper
19
20
21  def _return_none() -> None:
22      """Return None.
23
24      This function only exists to make the defaultdict in :class:`Session` pickle-able.
25      """
26      return None
27
28
29  class Session:
30      """Hold information about a session and provide methods to save and load that data."""
31
32      __slots__ = ('matrix_wrapper', 'polygon_points')
33
34      def __init__(
35          self,
36          *,
37          matrix_wrapper: MatrixWrapper,
38          polygon_points: List[Tuple[float, float]]
39      ) -> None:
40          """Create a :class:`Session` object with the given data."""
```

```
41              self.matrix_wrapper = matrix_wrapper
42              self.polygon_points = polygon_points
43
44          def save_to_file(self, filename: str) -> None:
45              """Save the session state to a file, creating parent directories as needed."""
46              parent_dir = pathlib.Path(os.path.expanduser(filename)).parent.absolute()
47
48              if not os.path.isdir(parent_dir):
49                  os.makedirs(parent_dir)
50
51              data_dict: DefaultDict[str, Any] = defaultdict(_return_none, lintrans=lintrans.__version__)
52              for attr in self.__slots__:
53                  data_dict[attr] = getattr(self, attr)
54
55              with open(filename, 'wb') as f:
56                  pickle.dump(data_dict, f, protocol=4)
57
58          @classmethod
59          def load_from_file(cls, filename: str) -> Tuple[Session, str, bool]:
60              """Return the session state that was previously saved to ``filename`` along with some extra information.
61
62              The tuple we return has the :class:`Session` object (with some possibly None arguments),
63              the lintrans version that the file was saved under, and whether the file had any extra
64              attributes that this version doesn't support.
65
66              :raises AttributeError: For specific older versions of :class:`Session` before it used ``__slots__``
67              :raises EOFError: If the file doesn't contain a pickled Python object
68              :raises FileNotFoundError: If the file doesn't exist
69              :raises ValueError: If the file contains a pickled object of the wrong type
70              """
71              with open(filename, 'rb') as f:
72                  data_dict = pickle.load(f)
73
74              if not isinstance(data_dict, defaultdict):
75                  raise ValueError(f'File {filename} contains pickled object of the wrong type (must be defaultdict)')
76
77              session = cls(
78                  matrix_wrapper=data_dict['matrix_wrapper'],
79                  polygon_points=data_dict['polygon_points']
80              )
81
82              # Check if the file has more attributes than we expect
83              # If it does, it's probably from a higher version of lintrans
84              extra_attrs = len(
85                  set(data_dict.keys()).difference(
86                      set(['lintrans', *cls.__slots__])
87                  )
88              ) != 0
89
90              return session, data_dict['lintrans'], extra_attrs
```

A.8 gui/settings.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module contains the :class:`DisplaySettings` class, which holds configuration for display."""
8
9   from __future__ import annotations
10
11  from dataclasses import dataclass
12
13
14  @dataclass
15  class DisplaySettings:
16      """This class simply holds some attributes to configure display."""
17
```

```
18          # === Basic stuff
19
20          draw_background_grid: bool = True
21          """This controls whether we want to draw the background grid.
22
23          The background axes will always be drawn. This makes it easy to identify the center of the space.
24          """
25
26          draw_transformed_grid: bool = True
27          """This controls whether we want to draw the transformed grid. Vectors are handled separately."""
28
29          draw_basis_vectors: bool = True
30          """This controls whether we want to draw the transformed basis vectors."""
31
32          label_basis_vectors: bool = False
33          """This controls whether we want to label the `i` and `j` basis vectors."""
34
35          # === Animations
36
37          smoothen_determinant: bool = True
38          """This controls whether we want the determinant to change smoothly during the animation.
39
40          .. note::
41             Even if this is ``True``, it will be ignored if we're animating from a positive det matrix to
42             a negative det matrix, or vice versa, because if we try to smoothly animate that determinant,
43             things blow up and the app often crashes.
44          """
45
46          applicative_animation: bool = True
47          """There are two types of simple animation, transitional and applicative.
48
49          Let ``C`` be the matrix representing the currently displayed transformation, and let ``T`` be the target matrix.
50          Transitional animation means that we animate directly from ``C`` from ``T``,
51          and applicative animation means that we animate from ``C`` to ``TC``, so we apply ``T`` to ``C``.
52          """
53
54          animation_time: int = 1200
55          """This is the number of milliseconds that an animation takes."""
56
57          animation_pause_length: int = 400
58          """This is the number of milliseconds that we wait between animations when using comma syntax."""
59
60          # === Matrix info
61
62          draw_determinant_parallelogram: bool = False
63          """This controls whether or not we should shade the parallelogram representing the determinant of the matrix."""
64
65          show_determinant_value: bool = True
66          """This controls whether we should write the text value of the determinant inside the parallelogram.
67
68          The text only gets draw if :attr:`draw_determinant_parallelogram` is also True.
69          """
70
71          draw_eigenvectors: bool = False
72          """This controls whether we should draw the eigenvectors of the transformation."""
73
74          draw_eigenlines: bool = False
75          """This controls whether we should draw the eigenlines of the transformation."""
76
77          # === Polygon
78
79          draw_untransformed_polygon: bool = True
80          """This controls whether we should draw the untransformed version of the user-defined polygon."""
81
82          draw_transformed_polygon: bool = True
83          """This controls whether we should draw the transformed version of the user-defined polygon."""
84
85          # === Input/output vectors
86
87          draw_io_vectors: bool = True
88          """This controls whether we should draw the input and output vectors in the main viewport."""
```

A.9 gui/utility.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This module provides utility functions for the whole GUI, such as :func:`qapp`."""

from PyQt5.QtCore import QCoreApplication


def qapp() -> QCoreApplication:
    """Return the equivalent of the global :class:`qApp` pointer.

    :raises RuntimeError: If :meth:`QCoreApplication.instance` returns ``None``
    """
    instance = QCoreApplication.instance()

    if instance is None:
        raise RuntimeError('qApp undefined')

    return instance
```

A.10 gui/__init__.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This package supplies the main GUI and associated dialogs for visualization."""

from . import dialogs, plots, session, settings, utility, validate
from .main_window import main

__all__ = ['dialogs', 'main', 'plots', 'session', 'settings', 'utility', 'validate']
```

A.11 gui/plots/classes.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This module provides superclasses for plotting transformations."""

from __future__ import annotations

from abc import abstractmethod
from math import ceil, dist, floor
from typing import Iterable, List, Optional, Tuple

import numpy as np
from PyQt5.QtCore import QPoint, QPointF, QRectF, Qt
from PyQt5.QtGui import (QBrush, QColor, QFont, QMouseEvent, QPainter, QPainterPath,
                         QPaintEvent, QPen, QPolygonF, QWheelEvent)
from PyQt5.QtWidgets import QWidget

from lintrans.typing_ import MatrixType, VectorType


class BackgroundPlot(QWidget):
    """This class provides a background for plotting, as well as setup for a Qt widget.
```

```
26
27          This class provides a background (untransformed) plane, and all the backend details
28          for a Qt application, but does not provide useful functionality. To be useful,
29          this class must be subclassed and behaviour must be implemented by the subclass.
30          """
31
32          DEFAULT_GRID_SPACING: int = 85
33          """This is the starting spacing between grid lines (in pixels)."""
34
35          _MINIMUM_GRID_SPACING: int = 5
36          """This is the minimum spacing between grid lines (in pixels)."""
37
38          _COLOUR_BACKGROUND_GRID: QColor = QColor('#808080')
39          """This is the colour of the background grid lines."""
40
41          _COLOUR_BACKGROUND_AXES: QColor = QColor('#000000')
42          """This is the colour of the background axes."""
43
44          _WIDTH_BACKGROUND_GRID: float = 0.3
45          """This is the width of the background grid lines, as a multiple of the :class:`QPainter` line width."""
46
47          _PEN_POLYGON: QPen = QPen(QColor('#000000'), 1.5)
48          """This is the pen used to draw the normal polygon."""
49
50          _BRUSH_SOLID_WHITE: QBrush = QBrush(QColor('#FFFFFF'), Qt.SolidPattern)
51          """This brush is just solid white. Used to draw the insides of circles."""
52
53          def __init__(self, *args, **kwargs):
54              """Create the widget and setup backend stuff for rendering.
55
56              .. note:: ``*args`` and ``**kwargs`` are passed the superclass constructor (:class:`QWidget`).
57              """
58              super().__init__(*args, **kwargs)
59
60              self.setAutoFillBackground(True)
61
62              # Set the background to white
63              palette = self.palette()
64              palette.setColor(self.backgroundRole(), Qt.white)
65              self.setPalette(palette)
66
67              self.grid_spacing = self.DEFAULT_GRID_SPACING
68
69          @property
70          def _canvas_origin(self) -> Tuple[int, int]:
71              """Return the canvas coords of the grid origin.
72
73              The return value is intended to be unpacked and passed to a :meth:`QPainter.drawLine:iiii` call.
74
75              See :meth:`canvas_coords`.
76
77              :returns: The canvas coordinates of the grid origin
78              :rtype: Tuple[int, int]
79              """
80              return self.width() // 2, self.height() // 2
81
82          def _canvas_x(self, x: float) -> int:
83              """Convert an x coordinate from grid coords to canvas coords."""
84              return int(self._canvas_origin[0] + x * self.grid_spacing)
85
86          def _canvas_y(self, y: float) -> int:
87              """Convert a y coordinate from grid coords to canvas coords."""
88              return int(self._canvas_origin[1] - y * self.grid_spacing)
89
90          def canvas_coords(self, x: float, y: float) -> Tuple[int, int]:
91              """Convert a coordinate from grid coords to canvas coords.
92
93              This method is intended to be used like
94
95              .. code::
96
97                 painter.drawLine(*self.canvas_coords(x1, y1), *self.canvas_coords(x2, y2))
98
```

```
            or like

            .. code::

               painter.drawLine(*self._canvas_origin, *self.canvas_coords(x, y))

            See :attr:`_canvas_origin`.

            :param float x: The x component of the grid coordinate
            :param float y: The y component of the grid coordinate
            :returns: The resultant canvas coordinates
            :rtype: Tuple[int, int]
            """
            return self._canvas_x(x), self._canvas_y(y)

        def _grid_corner(self) -> Tuple[float, float]:
            """Return the grid coords of the top right corner."""
            return self.width() / (2 * self.grid_spacing), self.height() / (2 * self.grid_spacing)

        def _grid_coords(self, x: int, y: int) -> Tuple[float, float]:
            """Convert a coordinate from canvas coords to grid coords.

            :param int x: The x component of the canvas coordinate
            :param int y: The y component of the canvas coordinate
            :returns: The resultant grid coordinates
            :rtype: Tuple[float, float]
            """
            # We get the maximum grid coords and convert them into canvas coords
            return (x - self._canvas_origin[0]) / self.grid_spacing, (-y + self._canvas_origin[1]) / self.grid_spacing

        @abstractmethod
        def paintEvent(self, event: QPaintEvent) -> None:
            """Handle a :class:`QPaintEvent`.

            .. note:: This method is abstract and must be overridden by all subclasses.
            """

        def _draw_background(self, painter: QPainter, draw_grid: bool) -> None:
            """Draw the background grid.

            .. note:: This method is just a utility method for subclasses to use to render the background grid.

            :param QPainter painter: The painter to draw the background with
            :param bool draw_grid: Whether to draw the grid lines
            """
            if draw_grid:
                painter.setPen(QPen(self._COLOUR_BACKGROUND_GRID, self._WIDTH_BACKGROUND_GRID))

                # Draw equally spaced vertical lines, starting in the middle and going out
                # We loop up to half of the width. This is because we draw a line on each side in each iteration
                for x in range(self.width() // 2 + self.grid_spacing, self.width(), self.grid_spacing):
                    painter.drawLine(x, 0, x, self.height())
                    painter.drawLine(self.width() - x, 0, self.width() - x, self.height())

                # Same with the horizontal lines
                for y in range(self.height() // 2 + self.grid_spacing, self.height(), self.grid_spacing):
                    painter.drawLine(0, y, self.width(), y)
                    painter.drawLine(0, self.height() - y, self.width(), self.height() - y)

            # Now draw the axes
            painter.setPen(QPen(self._COLOUR_BACKGROUND_AXES, self._WIDTH_BACKGROUND_GRID))
            painter.drawLine(self.width() // 2, 0, self.width() // 2, self.height())
            painter.drawLine(0, self.height() // 2, self.width(), self.height() // 2)

        def wheelEvent(self, event: QWheelEvent) -> None:
            """Handle a :class:`QWheelEvent` by zooming in or our of the grid."""
            # angleDelta() returns a number of units equal to 8 times the number of degrees rotated
            degrees = event.angleDelta() / 8

            if degrees is not None:
                new_spacing = max(1, self.grid_spacing + degrees.y())

                if new_spacing >= self._MINIMUM_GRID_SPACING:
```

```
                self.grid_spacing = new_spacing

        event.accept()
        self.update()


class InteractivePlot(BackgroundPlot):
    """This class represents an interactive plot, which allows the user to click and/or drag point(s).

    It declares the Qt methods needed for mouse cursor interaction to be abstract,
    requiring all subclasses to implement these.
    """

    _CURSOR_EPSILON: int = 5
    """This is the distance (in pixels) that the cursor needs to be from the point to drag it."""

    _SNAP_DIST = 0.1
    """This is the distance (in grid coords) that the cursor needs to be from an integer point to snap to it."""

    def _round_to_int_coord(self, point: Tuple[float, float]) -> Tuple[float, float]:
        """Take a coordinate in grid coords and round it to an integer coordinate if it's within :attr:`_SNAP_DIST`.

        If the point is not close enough, we just return the original point.
        """
        x, y = point

        possible_snaps: List[Tuple[int, int]] = [
            (floor(x), floor(y)),
            (floor(x), ceil(y)),
            (ceil(x), floor(y)),
            (ceil(x), ceil(y))
        ]

        snap_distances: List[Tuple[float, Tuple[int, int]]] = [
            (dist((x, y), coord), coord)
            for coord in possible_snaps
        ]

        for snap_dist, coord in snap_distances:
            if snap_dist < self._SNAP_DIST:
                x, y = coord

        return x, y

    def _is_within_epsilon(self, cursor_pos: Tuple[float, float], point: Tuple[float, float]) -> bool:
        """Check if the cursor position (in canvas coords) is within range of the given point."""
        mx, my = cursor_pos
        px, py = self.canvas_coords(*point)
        return (abs(px - mx) <= self._CURSOR_EPSILON and abs(py - my) <= self._CURSOR_EPSILON)

    @abstractmethod
    def mousePressEvent(self, event: QMouseEvent) -> None:
        """Handle the mouse being pressed."""

    @abstractmethod
    def mouseReleaseEvent(self, event: QMouseEvent) -> None:
        """Handle the mouse being released."""

    @abstractmethod
    def mouseMoveEvent(self, event: QMouseEvent) -> None:
        """Handle the mouse moving on the widget."""


class VectorGridPlot(BackgroundPlot):
    """This class represents a background plot, with vectors and their grid drawn on top. It provides utility
    ↪  methods.

    .. note::
        This is a simple superclass for vectors and is not for visualizing transformations.
        See :class:`VisualizeTransformationPlot`.

    This class should be subclassed to be used for visualization and matrix definition widgets.
    All useful behaviour should be implemented by any subclass.
```

```
244
245         .. warning:: This class should never be directly instantiated, only subclassed.
246         """
247
248         _COLOUR_I = QColor('#0808d8')
249         """This is the colour of the `i` basis vector and associated transformed grid lines."""
250
251         _COLOUR_J = QColor('#e90000')
252         """This is the colour of the `j` basis vector and associated transformed grid lines."""
253
254         _COLOUR_TEXT = QColor('#000000')
255         """This is the colour of the text."""
256
257         _WIDTH_VECTOR_LINE = 1.8
258         """This is the width of the transformed basis vector lines, as a multiple of the :class:`QPainter` line
        ↪  width."""
259
260         _WIDTH_TRANSFORMED_GRID = 0.8
261         """This is the width of the transformed grid lines, as a multiple of the :class:`QPainter` line width."""
262
263         _ARROWHEAD_LENGTH = 0.15
264         """This is the minimum length (in grid coord size) of the arrowhead parts."""
265
266         _MAX_PARALLEL_LINES = 150
267         """This is the maximum number of parallel transformed grid lines that will be drawn.
268
269         The user can zoom out further, but we will stop drawing grid lines beyond this number.
270         """
271
272         def __init__(self, *args, **kwargs):
273             """Create the widget with ``point_i`` and ``point_j`` attributes.
274
275             .. note:: ``*args`` and ``**kwargs`` are passed to the superclass constructor (:class:`BackgroundPlot`).
276             """
277             super().__init__(*args, **kwargs)
278
279             self.point_i: Tuple[float, float] = (1., 0.)
280             self.point_j: Tuple[float, float] = (0., 1.)
281
282         @property
283         def _matrix(self) -> MatrixType:
284             """Return the assembled matrix of the basis vectors."""
285             return np.array([
286                 [self.point_i[0], self.point_j[0]],
287                 [self.point_i[1], self.point_j[1]]
288             ])
289
290         @property
291         def _det(self) -> float:
292             """Return the determinant of the assembled matrix."""
293             return float(np.linalg.det(self._matrix))
294
295         @property
296         def _eigs(self) -> 'Iterable[Tuple[float, VectorType]]':
297             """Return the eigenvalues and eigenvectors zipped together to be iterated over.
298
299             :rtype: Iterable[Tuple[float, VectorType]]
300             """
301             values, vectors = np.linalg.eig(self._matrix)
302             return zip(values, vectors.T)
303
304         @abstractmethod
305         def paintEvent(self, event: QPaintEvent) -> None:
306             """Handle a :class:`QPaintEvent`."""
307
308         def _draw_parallel_lines(self, painter: QPainter, vector: Tuple[float, float], point: Tuple[float, float]) ->
        ↪  None:
309             """Draw a set of evenly spaced grid lines parallel to ``vector`` intersecting ``point``.
310
311             :param QPainter painter: The painter to draw the lines with
312             :param vector: The vector to draw the grid lines parallel to
313             :type vector: Tuple[float, float]
314             :param point: The point for the lines to intersect with
```

```
        :type point: Tuple[float, float]
        """
        max_x, max_y = self._grid_corner()
        vector_x, vector_y = vector
        point_x, point_y = point

        # If the determinant is 0
        if abs(vector_x * point_y - vector_y * point_x) < 1e-12:
            rank = np.linalg.matrix_rank(
                np.array([
                    [vector_x, point_x],
                    [vector_y, point_y]
                ])
            )

            # If the matrix is rank 1, then we can draw the column space line
            if rank == 1:
                # If the vector does not have a 0 x or y component, then we can just draw the line
                if abs(vector_x) > 1e-12 and abs(vector_y) > 1e-12:
                    self._draw_oblique_line(painter, vector_y / vector_x, 0)

                # Otherwise, we have to draw lines along the axes
                elif abs(vector_x) > 1e-12 and abs(vector_y) < 1e-12:
                    painter.drawLine(0, self.height() // 2, self.width(), self.height() // 2)

                elif abs(vector_x) < 1e-12 and abs(vector_y) > 1e-12:
                    painter.drawLine(self.width() // 2, 0, self.width() // 2, self.height())

                # If the vector is (0, 0), then don't draw a line for it
                else:
                    return

            # If the rank is 0, then we don't draw any lines
            else:
                return

        elif abs(vector_x) < 1e-12 and abs(vector_y) < 1e-12:
            # If both components of the vector are practically 0, then we can't render any grid lines
            return

        # Draw vertical lines
        elif abs(vector_x) < 1e-12:
            painter.drawLine(self._canvas_x(0), 0, self._canvas_x(0), self.height())

            for i in range(min(abs(int(max_x / point_x)), self._MAX_PARALLEL_LINES)):
                painter.drawLine(
                    self._canvas_x((i + 1) * point_x),
                    0,
                    self._canvas_x((i + 1) * point_x),
                    self.height()
                )
                painter.drawLine(
                    self._canvas_x(-1 * (i + 1) * point_x),
                    0,
                    self._canvas_x(-1 * (i + 1) * point_x),
                    self.height()
                )

        # Draw horizontal lines
        elif abs(vector_y) < 1e-12:
            painter.drawLine(0, self._canvas_y(0), self.width(), self._canvas_y(0))

            for i in range(min(abs(int(max_y / point_y)), self._MAX_PARALLEL_LINES)):
                painter.drawLine(
                    0,
                    self._canvas_y((i + 1) * point_y),
                    self.width(),
                    self._canvas_y((i + 1) * point_y)
                )
                painter.drawLine(
                    0,
                    self._canvas_y(-1 * (i + 1) * point_y),
                    self.width(),
```

```
                    self._canvas_y(-1 * (i + 1) * point_y)
                )

        # If the line is oblique, then we can use y = mx + c
        else:
            m = vector_y / vector_x
            c = point_y - m * point_x

            self._draw_oblique_line(painter, m, 0)

            # We don't want to overshoot the max number of parallel lines,
            # but we should also stop looping as soon as we can't draw any more lines
            for i in range(1, self._MAX_PARALLEL_LINES + 1):
                if not self._draw_pair_of_oblique_lines(painter, m, i * c):
                    break

    def _draw_pair_of_oblique_lines(self, painter: QPainter, m: float, c: float) -> bool:
        """Draw a pair of oblique lines, using the equation y = mx + c.

        This method just calls :meth:`_draw_oblique_line` with ``c`` and ``-c``,
        and returns True if either call returned True.

        :param QPainter painter: The painter to draw the vectors and grid lines with
        :param float m: The gradient of the lines to draw
        :param float c: The y-intercept of the lines to draw. We use the positive and negative versions
        :returns bool: Whether we were able to draw any lines on the canvas
        """
        return any([
            self._draw_oblique_line(painter, m, c),
            self._draw_oblique_line(painter, m, -c)
        ])

    def _draw_oblique_line(self, painter: QPainter, m: float, c: float) -> bool:
        """Draw an oblique line, using the equation y = mx + c.

        We only draw the part of the line that fits within the canvas, returning True if
        we were able to draw a line within the boundaries, and False if we couldn't draw a line

        :param QPainter painter: The painter to draw the vectors and grid lines with
        :param float m: The gradient of the line to draw
        :param float c: The y-intercept of the line to draw
        :returns bool: Whether we were able to draw a line on the canvas
        """
        max_x, max_y = self._grid_corner()

        # These variable names are shortened for convenience
        # myi is max_y_intersection, mmyi is minus_max_y_intersection, etc.
        myi = (max_y - c) / m
        mmyi = (-max_y - c) / m
        mxi = max_x * m + c
        mmxi = -max_x * m + c

        # The inner list here is a list of coords, or None
        # If an intersection fits within the bounds, then we keep its coord,
        # else it is None, and then gets discarded from the points list
        # By the end, points is a list of two coords, or an empty list
        points: List[Tuple[float, float]] = [
            x for x in [
                (myi, max_y) if -max_x < myi < max_x else None,
                (mmyi, -max_y) if -max_x < mmyi < max_x else None,
                (max_x, mxi) if -max_y < mxi < max_y else None,
                (-max_x, mmxi) if -max_y < mmxi < max_y else None
            ] if x is not None
        ]

        # If no intersections fit on the canvas
        if len(points) < 2:
            return False

        # If we can, then draw the line
        else:
            painter.drawLine(
                *self.canvas_coords(*points[0]),
```

```
461                 *self.canvas_coords(*points[1])
462             )
463             return True
464
465     def _draw_transformed_grid(self, painter: QPainter) -> None:
466         """Draw the transformed version of the grid, given by the basis vectors.
467
468         .. note:: This method draws the grid, but not the basis vectors. Use :meth:`_draw_basis_vectors` to draw
    ↪ them.
469
470         :param QPainter painter: The painter to draw the grid lines with
471         """
472         # Draw all the parallel lines
473         painter.setPen(QPen(self._COLOUR_I, self._WIDTH_TRANSFORMED_GRID))
474         self._draw_parallel_lines(painter, self.point_i, self.point_j)
475         painter.setPen(QPen(self._COLOUR_J, self._WIDTH_TRANSFORMED_GRID))
476         self._draw_parallel_lines(painter, self.point_j, self.point_i)
477
478     def _draw_arrowhead_away_from_origin(self, painter: QPainter, point: Tuple[float, float]) -> None:
479         """Draw an arrowhead at ``point``, pointing away from the origin.
480
481         :param QPainter painter: The painter to draw the arrowhead with
482         :param point: The point to draw the arrowhead at, given in grid coords
483         :type point: Tuple[float, float]
484         """
485         # This algorithm was adapted from a C# algorithm found at
486         # http://csharphelper.com/blog/2014/12/draw-lines-with-arrowheads-in-c/
487
488         # Get the x and y coords of the point, and then normalize them
489         # We have to normalize them, or else the size of the arrowhead will
490         # scale with the distance of the point from the origin
491         x, y = point
492         vector_length = np.sqrt(x * x + y * y)
493
494         if vector_length < 1e-12:
495             return
496
497         nx = x / vector_length
498         ny = y / vector_length
499
500         # We choose a length and find the steps in the x and y directions
501         length = min(
502             self._ARROWHEAD_LENGTH * self.DEFAULT_GRID_SPACING / self.grid_spacing,
503             vector_length
504         )
505         dx = length * (-nx - ny)
506         dy = length * (nx - ny)
507
508         # Then we just plot those lines
509         painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x + dx, y + dy))
510         painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x - dy, y + dx))
511
512     def _draw_position_vector(self, painter: QPainter, point: Tuple[float, float], colour: QColor) -> None:
513         """Draw a vector from the origin to the given point.
514
515         :param QPainter painter: The painter to draw the position vector with
516         :param point: The tip of the position vector in grid coords
517         :type point: Tuple[float, float]
518         :param QColor colour: The colour to draw the position vector in
519         """
520         painter.setPen(QPen(colour, self._WIDTH_VECTOR_LINE))
521         painter.drawLine(*self._canvas_origin, *self.canvas_coords(*point))
522         self._draw_arrowhead_away_from_origin(painter, point)
523
524     def _draw_basis_vectors(self, painter: QPainter) -> None:
525         """Draw arrowheads at the tips of the basis vectors.
526
527         :param QPainter painter: The painter to draw the basis vectors with
528         """
529         self._draw_position_vector(painter, self.point_i, self._COLOUR_I)
530         self._draw_position_vector(painter, self.point_j, self._COLOUR_J)
531
532     def _draw_basis_vector_labels(self, painter: QPainter) -> None:
```

```
533             """Label the basis vectors with `i` and `j`."""
534             font = self.font()
535             font.setItalic(True)
536             font.setStyleHint(QFont.Serif)
537 
538             self._draw_text_at_vector_tip(painter, self.point_i, 'i', font)
539             self._draw_text_at_vector_tip(painter, self.point_j, 'j', font)
540 
541         def _draw_text_at_vector_tip(
542             self,
543             painter: QPainter,
544             point: Tuple[float, float],
545             text: str,
546             font: Optional[QFont] = None
547         ) -> None:
548             """Draw the given text at the point as if it were the tip of a vector, using the custom font if given."""
549             offset = 3
550             top_left: QPoint
551             bottom_right: QPoint
552             alignment_flags: int
553             x, y = point
554 
555             if x >= 0 and y >= 0:  # Q1
556                 top_left = QPoint(self._canvas_x(x) + offset, 0)
557                 bottom_right = QPoint(self.width(), self._canvas_y(y) - offset)
558                 alignment_flags = Qt.AlignLeft | Qt.AlignBottom
559 
560             elif x < 0 and y >= 0:  # Q2
561                 top_left = QPoint(0, 0)
562                 bottom_right = QPoint(self._canvas_x(x) - offset, self._canvas_y(y) - offset)
563                 alignment_flags = Qt.AlignRight | Qt.AlignBottom
564 
565             elif x < 0 and y < 0:  # Q3
566                 top_left = QPoint(0, self._canvas_y(y) + offset)
567                 bottom_right = QPoint(self._canvas_x(x) - offset, self.height())
568                 alignment_flags = Qt.AlignRight | Qt.AlignTop
569 
570             else:  # Q4
571                 top_left = QPoint(self._canvas_x(x) + offset, self._canvas_y(y) + offset)
572                 bottom_right = QPoint(self.width(), self.height())
573                 alignment_flags = Qt.AlignLeft | Qt.AlignTop
574 
575             original_font = painter.font()
576 
577             if font is not None:
578                 painter.setFont(font)
579 
580             painter.setPen(QPen(self._COLOUR_TEXT, 1))
581             painter.drawText(QRectF(top_left, bottom_right), alignment_flags, text)
582 
583             painter.setFont(original_font)
584 
585 
586     class VisualizeTransformationPlot(VectorGridPlot):
587         """This class is a superclass for visualizing transformations. It provides utility methods."""
588 
589         _COLOUR_EIGEN = QColor('#13cf00')
590         """This is the colour of the eigenvectors and eigenlines (the spans of the eigenvectors)."""
591 
592         @abstractmethod
593         def paintEvent(self, event: QPaintEvent) -> None:
594             """Handle a :class:`QPaintEvent`."""
595 
596         def _draw_determinant_parallelogram(self, painter: QPainter) -> None:
597             """Draw the parallelogram of the determinant of the matrix.
598 
599             :param QPainter painter: The painter to draw the parallelogram with
600             """
601             if self._det == 0:
602                 return
603 
604             path = QPainterPath()
605             path.moveTo(*self._canvas_origin)
```

```
606             path.lineTo(*self.canvas_coords(*self.point_i))
607             path.lineTo(*self.canvas_coords(self.point_i[0] + self.point_j[0], self.point_i[1] + self.point_j[1]))
608             path.lineTo(*self.canvas_coords(*self.point_j))
609 
610             color = (16, 235, 253) if self._det > 0 else (253, 34, 16)
611             brush = QBrush(QColor(*color, alpha=128), Qt.SolidPattern)
612 
613             painter.fillPath(path, brush)
614 
615         def _draw_determinant_text(self, painter: QPainter) -> None:
616             """Write the string value of the determinant in the middle of the parallelogram.
617 
618             :param QPainter painter: The painter to draw the determinant text with
619             """
620             painter.setPen(QPen(self._COLOUR_TEXT, self._WIDTH_VECTOR_LINE))
621 
622             # We're building a QRect that encloses the determinant parallelogram
623             # Then we can center the text in this QRect
624             coords: List[Tuple[float, float]] = [
625                 (0, 0),
626                 self.point_i,
627                 self.point_j,
628                 (
629                     self.point_i[0] + self.point_j[0],
630                     self.point_i[1] + self.point_j[1]
631                 )
632             ]
633 
634             xs = [t[0] for t in coords]
635             ys = [t[1] for t in coords]
636 
637             top_left = QPoint(*self.canvas_coords(min(xs), max(ys)))
638             bottom_right = QPoint(*self.canvas_coords(max(xs), min(ys)))
639 
640             rect = QRectF(top_left, bottom_right)
641 
642             painter.drawText(
643                 rect,
644                 Qt.AlignHCenter | Qt.AlignVCenter,
645                 f'{self._det:.2f}'
646             )
647 
648         def _draw_eigenvectors(self, painter: QPainter) -> None:
649             """Draw the eigenvectors of the displayed matrix transformation.
650 
651             :param QPainter painter: The painter to draw the eigenvectors with
652             """
653             for value, vector in self._eigs:
654                 x = value * vector[0]
655                 y = value * vector[1]
656 
657                 if x.imag != 0 or y.imag != 0:
658                     continue
659 
660                 self._draw_position_vector(painter, (x, y), self._COLOUR_EIGEN)
661                 self._draw_text_at_vector_tip(painter, (x, y), f'{value:.2f}')
662 
663         def _draw_eigenlines(self, painter: QPainter) -> None:
664             """Draw the eigenlines. These are the invariant lines, or the spans of the eigenvectors.
665 
666             :param QPainter painter: The painter to draw the eigenlines with
667             """
668             painter.setPen(QPen(self._COLOUR_EIGEN, self._WIDTH_TRANSFORMED_GRID))
669 
670             for value, vector in self._eigs:
671                 if value.imag != 0:
672                     continue
673 
674                 x, y = vector
675 
676                 if x == 0:
677                     x_mid = int(self.width() / 2)
678                     painter.drawLine(x_mid, 0, x_mid, self.height())
```

```
679
680             elif y == 0:
681                 y_mid = int(self.height() / 2)
682                 painter.drawLine(0, y_mid, self.width(), y_mid)
683
684             else:
685                 self._draw_oblique_line(painter, y / x, 0)
686
687     def _draw_polygon_from_points(self, painter: QPainter, points: List[Tuple[float, float]]) -> None:
688         """Draw a polygon from a given list of points.
689
690         This is a helper method for :meth:`_draw_untransformed_polygon` and :meth:`_draw_transformed_polygon`.
691         """
692         if len(points) > 2:
693             painter.drawPolygon(QPolygonF(
694                 [QPointF(*self.canvas_coords(*p)) for p in points]
695             ))
696         elif len(points) == 2:
697             painter.drawLine(
698                 *self.canvas_coords(*points[0]),
699                 *self.canvas_coords(*points[1])
700             )
701
702     def _draw_untransformed_polygon(self, painter: QPainter) -> None:
703         """Draw the original untransformed polygon with a dashed line."""
704         pen = QPen(self._PEN_POLYGON)
705         pen.setDashPattern([4, 4])
706         painter.setPen(pen)
707
708         self._draw_polygon_from_points(painter, self.polygon_points)
709
710     def _draw_transformed_polygon(self, painter: QPainter) -> None:
711         """Draw the transformed version of the polygon."""
712         if len(self.polygon_points) == 0:
713             return
714
715         painter.setPen(self._PEN_POLYGON)
716
717         # This transpose trick lets us do one matrix multiplication to transform every point in the polygon
718         # I learned this from Phil. Thanks Phil
719         self._draw_polygon_from_points(
720             painter,
721             (self._matrix @ np.array(self.polygon_points).T).T
722         )
```

A.12 gui/plots/widgets.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides the actual widgets that can be used to visualize transformations in the GUI."""
8
9   from __future__ import annotations
10
11  import operator
12  from abc import abstractmethod
13  from math import dist
14  from typing import List, Optional, Tuple
15
16  from PyQt5.QtCore import Qt, QPointF, pyqtSlot
17  from PyQt5.QtGui import QBrush, QColor, QMouseEvent, QPainter, QPaintEvent, QPen, QPolygonF
18
19  from lintrans.typing_ import MatrixType
20  from lintrans.gui.settings import DisplaySettings
21  from .classes import InteractivePlot, VisualizeTransformationPlot
22
23
```

```
24  class VisualizeTransformationWidget(VisualizeTransformationPlot):
25      """This widget is used in the main window to visualize transformations.
26
27      It handles all the rendering itself, and the only method that the user needs to care about
28      is :meth:`plot_matrix`, which allows you to visualize the given matrix transformation.
29      """
30
31      def __init__(self, *args, display_settings: DisplaySettings, polygon_points: List[Tuple[float, float]],
        ↪  **kwargs):
32          """Create the widget and assign its display settings, passing ``*args`` and ``**kwargs`` to super."""
33          super().__init__(*args, **kwargs)
34
35          self.display_settings = display_settings
36          self.polygon_points = polygon_points
37
38      def plot_matrix(self, matrix: MatrixType) -> None:
39          """Plot the given matrix on the grid by setting the basis vectors.
40
41          .. warning:: This method does not call :meth:`QWidget.update()`. This must be done by the caller.
42
43          :param MatrixType matrix: The matrix to plot
44          """
45          self.point_i = (matrix[0][0], matrix[1][0])
46          self.point_j = (matrix[0][1], matrix[1][1])
47
48      def _draw_scene(self, painter: QPainter) -> None:
49          """Draw the default scene of the transformation.
50
51          This method exists to make it easier to split the main viewport from visual definitions while
52          not using multiple :class:`QPainter` objects from a single :meth:`paintEvent` call in a subclass.
53          """
54          painter.setRenderHint(QPainter.Antialiasing)
55          painter.setBrush(Qt.NoBrush)
56
57          self._draw_background(painter, self.display_settings.draw_background_grid)
58
59          if self.display_settings.draw_eigenlines:
60              self._draw_eigenlines(painter)
61
62          if self.display_settings.draw_eigenvectors:
63              self._draw_eigenvectors(painter)
64
65          if self.display_settings.draw_determinant_parallelogram:
66              self._draw_determinant_parallelogram(painter)
67
68              if self.display_settings.show_determinant_value:
69                  self._draw_determinant_text(painter)
70
71          if self.display_settings.draw_transformed_grid:
72              self._draw_transformed_grid(painter)
73
74          if self.display_settings.draw_basis_vectors:
75              self._draw_basis_vectors(painter)
76
77              if self.display_settings.label_basis_vectors:
78                  self._draw_basis_vector_labels(painter)
79
80          if self.display_settings.draw_untransformed_polygon:
81              self._draw_untransformed_polygon(painter)
82
83          if self.display_settings.draw_transformed_polygon:
84              self._draw_transformed_polygon(painter)
85
86      @abstractmethod
87      def paintEvent(self, event: QPaintEvent) -> None:
88          """Paint the scene of the transformation."""
89
90
91  class MainViewportWidget(VisualizeTransformationWidget, InteractivePlot):
92      """This is the widget for the main viewport.
93
94      It extends :class:`VisualizeTransformationWidget` with input and output vectors.
95      """
```

```

    _COLOUR_OUTPUT_VECTOR = QColor('#f7c216')

    def __init__(self, *args, **kwargs):
        """Create the main viewport widget with its input point."""
        super().__init__(*args,  **kwargs)

        self._point_input: Tuple[float, float] = (1, 1)
        self._dragging_vector: bool = False

    def _draw_input_vector(self, painter: QPainter) -> None:
        """Draw the input vector."""
        pen = QPen(QColor('#000000'), self._WIDTH_VECTOR_LINE)
        painter.setPen(pen)

        x, y = self.canvas_coords(*self._point_input)
        painter.drawLine(*self._canvas_origin, x, y)

        painter.setBrush(self._BRUSH_SOLID_WHITE)

        painter.setPen(Qt.NoPen)
        painter.drawPie(
            x - self._CURSOR_EPSILON,
            y - self._CURSOR_EPSILON,
            2 * self._CURSOR_EPSILON,
            2 * self._CURSOR_EPSILON,
            0,
            16 * 360
        )

        painter.setPen(pen)
        painter.drawArc(
            x - self._CURSOR_EPSILON,
            y - self._CURSOR_EPSILON,
            2 * self._CURSOR_EPSILON,
            2 * self._CURSOR_EPSILON,
            0,
            16 * 360
        )

    def _draw_output_vector(self, painter: QPainter) -> None:
        """Draw the output vector."""
        painter.setPen(QPen(self._COLOUR_OUTPUT_VECTOR, self._WIDTH_VECTOR_LINE))
        painter.setBrush(QBrush(self._COLOUR_OUTPUT_VECTOR, Qt.SolidPattern))

        x, y = self.canvas_coords(*(self._matrix @ self._point_input))

        painter.drawLine(*self._canvas_origin, x, y)
        painter.drawPie(
            x - self._CURSOR_EPSILON,
            y - self._CURSOR_EPSILON,
            2 * self._CURSOR_EPSILON,
            2 * self._CURSOR_EPSILON,
            0,
            16 * 360
        )

    def paintEvent(self, event: QPaintEvent) -> None:
        """Paint the scene by just calling :meth:`_draw_scene` and drawing the I/O vectors."""
        painter = QPainter()
        painter.begin(self)

        self._draw_scene(painter)

        if self.display_settings.draw_io_vectors:
            self._draw_output_vector(painter)
            self._draw_input_vector(painter)

        painter.end()
        event.accept()

    def mousePressEvent(self, event: QMouseEvent) -> None:
        """Check if the user has clicked on the input vector."""
```

```
169             cursor_pos = (event.x(), event.y())
170
171             if event.button() != Qt.LeftButton:
172                 event.ignore()
173                 return
174
175             if self._is_within_epsilon(cursor_pos, self._point_input):
176                 self._dragging_vector = True
177
178             event.accept()
179
180         def mouseReleaseEvent(self, event: QMouseEvent) -> None:
181             """Stop dragging the input vector."""
182             if event.button() == Qt.LeftButton:
183                 self._dragging_vector = False
184                 event.accept()
185             else:
186                 event.ignore()
187
188         def mouseMoveEvent(self, event: QMouseEvent) -> None:
189             """Drag the input vector if the user has clicked on it."""
190             if not self._dragging_vector:
191                 event.ignore()
192                 return
193
194             x, y = self._round_to_int_coord(self._grid_coords(event.x(), event.y()))
195             self._point_input = (x, y)
196
197             self.update()
198             event.accept()
199
200
201     class DefineMatrixVisuallyWidget(VisualizeTransformationWidget, InteractivePlot):
202         """This widget allows the user to visually define a matrix.
203
204         This is just the widget itself. If you want the dialog, use
205         :class:`~lintrans.gui.dialogs.define_new_matrix.DefineVisuallyDialog`.
206         """
207
208         def __init__(self, *args, display_settings: DisplaySettings, polygon_points: List[Tuple[float, float]],
        ↪  **kwargs):
209             """Create the widget and enable mouse tracking. ``*args`` and ``**kwargs`` are passed to ``super()``."""
210             super().__init__(*args, display_settings=display_settings, polygon_points=polygon_points, **kwargs)
211
212             self._dragged_point: Tuple[float, float] | None = None
213
214         def paintEvent(self, event: QPaintEvent) -> None:
215             """Paint the scene by just calling :meth:`_draw_scene`."""
216             painter = QPainter()
217             painter.begin(self)
218
219             self._draw_scene(painter)
220
221             painter.end()
222             event.accept()
223
224         def mousePressEvent(self, event: QMouseEvent) -> None:
225             """Set the dragged point if the cursor is within :attr:`_CURSOR_EPSILON`."""
226             cursor_pos = (event.x(), event.y())
227
228             if event.button() != Qt.LeftButton:
229                 event.ignore()
230                 return
231
232             for point in (self.point_i, self.point_j):
233                 if self._is_within_epsilon(cursor_pos, point):
234                     self._dragged_point = point[0], point[1]
235
236             event.accept()
237
238         def mouseReleaseEvent(self, event: QMouseEvent) -> None:
239             """Handle the mouse click being released by unsetting the dragged point."""
240             if event.button() == Qt.LeftButton:
```

```
241                 self._dragged_point = None
242                 event.accept()
243             else:
244                 event.ignore()
245 
246         def mouseMoveEvent(self, event: QMouseEvent) -> None:
247             """Handle the mouse moving on the canvas."""
248             if self._dragged_point is None:
249                 event.ignore()
250                 return
251 
252             x, y = self._round_to_int_coord(self._grid_coords(event.x(), event.y()))
253 
254             if self._dragged_point == self.point_i:
255                 self.point_i = x, y
256 
257             elif self._dragged_point == self.point_j:
258                 self.point_j = x, y
259 
260             self._dragged_point = x, y
261 
262             self.update()
263             event.accept()
264 
265 
266     class DefinePolygonWidget(InteractivePlot):
267         """This widget allows the user to define a polygon by clicking and dragging points on the canvas."""
268 
269         def __init__(self, *args, polygon_points: List[Tuple[float, float]], **kwargs):
270             """Create the widget with a list of points and a dragged point index."""
271             super().__init__(*args, **kwargs)
272 
273             self._dragged_point_index: Optional[int] = None
274             self.points = polygon_points.copy()
275 
276         @pyqtSlot()
277         def reset_polygon(self) -> None:
278             """Reset the polygon and update the widget."""
279             self.points = []
280             self.update()
281 
282         def mousePressEvent(self, event: QMouseEvent) -> None:
283             """Handle the mouse being clicked by adding a point or setting the dragged point index to an existing
                ↪  point."""
284             if event.button() not in (Qt.LeftButton, Qt.RightButton):
285                 event.ignore()
286                 return
287 
288             canvas_pos = (event.x(), event.y())
289             grid_pos = self._grid_coords(*canvas_pos)
290 
291             if event.button() == Qt.LeftButton:
292                 for i, point in enumerate(self.points):
293                     if self._is_within_epsilon(canvas_pos, point):
294                         self._dragged_point_index = i
295                         event.accept()
296                         return
297 
298                 new_point = self._round_to_int_coord(grid_pos)
299 
300                 if len(self.points) < 2:
301                     self.points.append(new_point)
302                     self._dragged_point_index = -1
303                 else:
304                     # FIXME: This algorithm doesn't work very well when the new point is far away
305                     # from the existing polygon; it just picks the longest side
306 
307                     # Get a list of line segments and a list of their lengths
308                     line_segments = list(zip(self.points, self.points[1:])) + [(self.points[-1], self.points[0])]
309                     segment_lengths = map(lambda t: dist(*t), line_segments)
310 
311                     # Get the distance from each point in the polygon to the new point
312                     distances_to_point = [dist(p, new_point) for p in self.points]
```

```

                # For each pair of list-adjacent points, zip their distances to
                # the new point into a tuple, and add them together
                # This gives us the lengths of the catheti of the triangles that
                # connect the new point to each pair of adjacent points
                dist_to_point_pairs = list(zip(distances_to_point, distances_to_point[1:])) + \
                    [(distances_to_point[-1], distances_to_point[0])]

                # mypy doesn't like the use of sum for some reason. Just ignore it
                point_triangle_lengths = map(sum, dist_to_point_pairs)  # type: ignore[arg-type]

                # The normalized distance is the sum of the distances to the ends of the line segment
                # (point_triangle_lengths) divided by the length of the segment
                normalized_distances = list(map(operator.truediv, point_triangle_lengths, segment_lengths))

                # Get the best distance and insert this new point just after the point with that index
                # This will put it in the middle of the closest line segment
                best_distance = min(normalized_distances)
                index = 1 + normalized_distances.index(best_distance)

                self.points.insert(index, new_point)
                self._dragged_point_index = index

        elif event.button() == Qt.RightButton:
            for i, point in enumerate(self.points):
                if self._is_within_epsilon(canvas_pos, point):
                    self.points.pop(i)
                    break

        self.update()
        event.accept()

    def mouseReleaseEvent(self, event: QMouseEvent) -> None:
        """Handle the mouse click being released by unsetting the dragged point index."""
        if event.button() == Qt.LeftButton:
            self._dragged_point_index = None
            event.accept()
        else:
            event.ignore()

    def mouseMoveEvent(self, event: QMouseEvent) -> None:
        """Handle mouse movement by dragging the selected point."""
        if self._dragged_point_index is None:
            event.ignore()
            return

        x, y = self._round_to_int_coord(self._grid_coords(event.x(), event.y()))

        self.points[self._dragged_point_index] = x, y

        self.update()

        event.accept()

    def _draw_polygon(self, painter: QPainter) -> None:
        """Draw the polygon with circles at its vertices."""
        painter.setPen(self._PEN_POLYGON)

        if len(self.points) > 2:
            painter.drawPolygon(QPolygonF(
                [QPointF(*self.canvas_coords(*p)) for p in self.points]
            ))
        elif len(self.points) == 2:
            painter.drawLine(
                *self.canvas_coords(*self.points[0]),
                *self.canvas_coords(*self.points[1])
            )

        painter.setBrush(self._BRUSH_SOLID_WHITE)

        for point in self.points:
            x, y = self.canvas_coords(*point)

```

```
386                 painter.setPen(Qt.NoPen)
387                 painter.drawPie(
388                     x - self._CURSOR_EPSILON,
389                     y - self._CURSOR_EPSILON,
390                     2 * self._CURSOR_EPSILON,
391                     2 * self._CURSOR_EPSILON,
392                     0,
393                     16 * 360
394                 )
395
396                 painter.setPen(self._PEN_POLYGON)
397                 painter.drawArc(
398                     x - self._CURSOR_EPSILON,
399                     y - self._CURSOR_EPSILON,
400                     2 * self._CURSOR_EPSILON,
401                     2 * self._CURSOR_EPSILON,
402                     0,
403                     16 * 360
404                 )
405
406             painter.setBrush(Qt.NoBrush)
407
408         def paintEvent(self, event: QPaintEvent) -> None:
409             """Draw the polygon on the canvas."""
410             painter = QPainter()
411             painter.begin(self)
412
413             painter.setRenderHint(QPainter.Antialiasing)
414             painter.setBrush(Qt.NoBrush)
415
416             self._draw_background(painter, True)
417
418             self._draw_polygon(painter)
419
420             painter.end()
421             event.accept()
```

A.13 gui/plots/__init__.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This package provides widgets for the visualization plot in the main window and the visual definition dialog."""
8
9   from .classes import BackgroundPlot, VectorGridPlot, VisualizeTransformationPlot
10  from .widgets import DefinePolygonWidget, DefineMatrixVisuallyWidget, MainViewportWidget,
    ↪  VisualizeTransformationWidget
11
12  __all__ = ['BackgroundPlot', 'DefinePolygonWidget', 'DefineMatrixVisuallyWidget', 'MainViewportWidget',
13             'VectorGridPlot', 'VisualizeTransformationPlot', 'VisualizeTransformationWidget']
```

A.14 gui/dialogs/misc.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides miscellaneous dialog classes like :class:`AboutDialog`."""
8
9   from __future__ import annotations
10
11  import os
12  import platform
```

```
13  from typing import List, Tuple, Union
14
15  from PyQt5.QtCore import PYQT_VERSION_STR, QT_VERSION_STR, Qt, pyqtSlot
16  from PyQt5.QtGui import QKeySequence
17  from PyQt5.QtWidgets import (QDialog, QFileDialog, QGridLayout, QHBoxLayout, QLabel, QPushButton,
18                               QShortcut, QSizePolicy, QSpacerItem, QVBoxLayout, QWidget)
19
20  import lintrans
21  from lintrans.gui.plots import DefinePolygonWidget
22  from lintrans.matrices import MatrixWrapper
23  from lintrans.matrices.utility import round_float
24  from lintrans.typing_ import MatrixType, is_matrix_type
25
26
27  class FixedSizeDialog(QDialog):
28      """A simple superclass to create modal dialog boxes with fixed size.
29
30      We override the :meth:`open` method to set the fixed size as soon as the dialog is opened modally.
31      """
32
33      def __init__(self, *args, **kwargs) -> None:
34          """Set the :cpp:enum:`Qt::WA_DeleteOnClose` attribute to ensure deletion of dialog."""
35          super().__init__(*args, **kwargs)
36          self.setAttribute(Qt.WA_DeleteOnClose)
37          self.setWindowFlag(Qt.WindowContextHelpButtonHint, False)
38
39      def open(self) -> None:
40          """Override :meth:`QDialog.open` to set the dialog to a fixed size."""
41          super().open()
42          self.setFixedSize(self.size())
43
44
45  class AboutDialog(FixedSizeDialog):
46      """A simple dialog class to display information about the app to the user.
47
48      It only has an :meth:`__init__` method because it only has label widgets, so no other methods are necessary
    ↪ here.
49      """
50
51      def __init__(self, *args, **kwargs):
52          """Create an :class:`AboutDialog` object with all the label widgets."""
53          super().__init__(*args, **kwargs)
54
55          self.setWindowTitle('About lintrans')
56
57          # === Create the widgets
58
59          label_title = QLabel(self)
60          label_title.setText(f'lintrans (version {lintrans.__version__})')
61          label_title.setAlignment(Qt.AlignCenter)
62
63          font_title = label_title.font()
64          font_title.setPointSize(font_title.pointSize() * 2)
65          label_title.setFont(font_title)
66
67          label_version_info = QLabel(self)
68          label_version_info.setText(
69              f'With Python version {platform.python_version()}\n'
70              f'Qt version {QT_VERSION_STR} and PyQt5 version {PYQT_VERSION_STR}\n'
71              f'Running on {platform.platform()}'
72          )
73          label_version_info.setAlignment(Qt.AlignCenter)
74
75          label_info = QLabel(self)
76          label_info.setText(
77              'lintrans is a program designed to help visualise<br>'
78              '2D linear transformations represented with matrices.<br><br>'
79              "It's designed for teachers and students and all feedback<br>"
80              'is greatly appreciated. Go to <em>Help</em> &gt; <em>Give feedback</em><br>'
81              'to report a bug or suggest a new feature, or you can<br>email me directly at '
82              '<a href="mailto:dyson.dyson@icloud.com" style="color: black;">dyson.dyson@icloud.com</a>.'
83          )
84          label_info.setAlignment(Qt.AlignCenter)
```

```
85            label_info.setTextFormat(Qt.RichText)
86            label_info.setOpenExternalLinks(True)
87
88            label_copyright = QLabel(self)
89            label_copyright.setText(
90                'This program is free software.<br>Copyright 2021-2022 D. Dyson (DoctorDalek1963).<br>'
91                'This program is licensed under GPLv3, which can be found '
92                '<a href="https://www.gnu.org/licenses/gpl-3.0.html" style="color: black;">here</a>.'
93            )
94            label_copyright.setAlignment(Qt.AlignCenter)
95            label_copyright.setTextFormat(Qt.RichText)
96            label_copyright.setOpenExternalLinks(True)
97
98            # === Arrange the widgets
99
100           self.setContentsMargins(10, 10, 10, 10)
101
102           vlay = QVBoxLayout()
103           vlay.setSpacing(20)
104           vlay.addWidget(label_title)
105           vlay.addWidget(label_version_info)
106           vlay.addWidget(label_info)
107           vlay.addWidget(label_copyright)
108
109           self.setLayout(vlay)
110
111
112   class InfoPanelDialog(FixedSizeDialog):
113       """A simple dialog class to display an info panel that shows all currently defined matrices."""
114
115       def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
116           """Create the dialog box with all the widgets needed to show the information."""
117           super().__init__(*args, **kwargs)
118
119           self.setWindowTitle('Defined matrices')
120
121           grid_layout = QGridLayout()
122           grid_layout.setSpacing(20)
123
124           bold_font = self.font()
125           bold_font.setBold(True)
126
127           name_value_pair: tuple[str, Union[MatrixType, str]]
128
129           # Each defined matrix will get a widget group. Each group will be a label for the name,
130           # a label for '=', and a container widget to either show the matrix numerically, or to
131           # show the expression that it's defined as
132           for i, name_value_pair in enumerate(matrix_wrapper.get_defined_matrices()):
133               name, value = name_value_pair
134
135               # Create all the widgets first
136               label_name = QLabel(self)
137               label_name.setText(name)
138               label_name.setFont(bold_font)
139
140               label_equals = QLabel(self)
141               label_equals.setText('=')
142
143               widget_matrix = self._get_matrix_widget(value)
144
145               # We want columns of at most 6 widget groups
146               # This column variable manages which column of defined matrices we're on
147               # It's multiplied by 3 because all the widgets are in a single grid layout
148               # I could factor out each triplet of widgets for a defined matrix into a container widget,
149               # but I prefer to keep the widget count lower to reduce any possible lag
150               column = 3 * (i // 6)
151
152               grid_layout.addWidget(
153                   label_name,
154                   i - 2 * column,
155                   column,
156                   Qt.AlignCenter
157               )
```

```
158                 grid_layout.addWidget(
159                     label_equals,
160                     i - 2 * column,
161                     column + 1,
162                     Qt.AlignCenter
163                 )
164                 grid_layout.addWidget(
165                     widget_matrix,
166                     i - 2 * column,
167                     column + 2,
168                     Qt.AlignCenter
169                 )
170 
171             self.setContentsMargins(10, 10, 10, 10)
172             self.setLayout(grid_layout)
173 
174         def _get_matrix_widget(self, matrix: Union[MatrixType, str]) -> QWidget:
175             """Return a :class:`QWidget` containing the value of the matrix.
176 
177             If the matrix is defined as an expression, it will be a simple :class:`QLabel`.
178             If the matrix is defined as a matrix, it will be a :class:`QWidget` container
179             with multiple :class:`QLabel` objects in it.
180             """
181             if isinstance(matrix, str):
182                 label = QLabel(self)
183                 label.setText(matrix)
184                 return label
185 
186             elif is_matrix_type(matrix):
187                 # tl = top left, br = bottom right, etc.
188                 label_tl = QLabel(self)
189                 label_tl.setText(round_float(matrix[0][0]))
190 
191                 label_tr = QLabel(self)
192                 label_tr.setText(round_float(matrix[0][1]))
193 
194                 label_bl = QLabel(self)
195                 label_bl.setText(round_float(matrix[1][0]))
196 
197                 label_br = QLabel(self)
198                 label_br.setText(round_float(matrix[1][1]))
199 
200                 # The parens need to be bigger than the numbers, but increasing the font size also
201                 # makes the font thicker, so we have to reduce the font weight by the same factor
202                 font_parens = self.font()
203                 font_parens.setPointSize(int(font_parens.pointSize() * 2.5))
204                 font_parens.setWeight(int(font_parens.weight() / 2.5))
205 
206                 label_paren_left = QLabel(self)
207                 label_paren_left.setText('(')
208                 label_paren_left.setFont(font_parens)
209 
210                 label_paren_right = QLabel(self)
211                 label_paren_right.setText(')')
212                 label_paren_right.setFont(font_parens)
213 
214                 container = QWidget(self)
215                 grid_layout = QGridLayout()
216 
217                 grid_layout.addWidget(label_paren_left, 0, 0, -1, 1)
218                 grid_layout.addWidget(label_tl, 0, 1)
219                 grid_layout.addWidget(label_tr, 0, 2)
220                 grid_layout.addWidget(label_bl, 1, 1)
221                 grid_layout.addWidget(label_br, 1, 2)
222                 grid_layout.addWidget(label_paren_right, 0, 3, -1, 1)
223 
224                 container.setLayout(grid_layout)
225 
226                 return container
227 
228             raise ValueError('Matrix was not MatrixType or str')
229 
230 
```

```
231     class FileSelectDialog(QFileDialog):
232         """A subclass of :class:`QFileDialog` that fixes an issue with the default suffix on UNIX platforms."""
233
234         def selectedFiles(self) -> List[str]:
235             """Return a list of strings containing the absolute paths of the selected files in the dialog.
236
237             There is an issue on UNIX platforms where a hidden directory will be recognised as a suffix.
238             For example, ``/home/dyson/.lintrans/saves/test`` should have ``.lt`` appended, but
239             ``.lintrans/saves/test`` gets recognised as the suffix, so the default suffix is not added.
240
241             To fix this, we just look at the basename and see if it needs a suffix added. We do this for
242             every name in the list, but there should be just one name, since this class is only intended
243             to be used for saving files. We still return the full list of filenames.
244             """
245             selected_files: List[str] = []
246
247             for filename in super().selectedFiles():
248                 # path will be the full path of the file, without the extension
249                 # This method understands hidden directories on UNIX platforms
250                 path, ext = os.path.splitext(filename)
251
252                 if ext == '':
253                     ext = '.' + self.defaultSuffix()
254
255                 selected_files.append(''.join((path, ext)))
256
257             return selected_files
258
259
260     class DefinePolygonDialog(FixedSizeDialog):
261         """This dialog class allows the use to define a polygon with :class:`DefinePolygonWidget`."""
262
263         def __init__(self, *args, polygon_points: List[Tuple[float, float]], **kwargs) -> None:
264             """Create the dialog with the :class:`DefinePolygonWidget` widget."""
265             super().__init__(*args, **kwargs)
266
267             self.setWindowTitle('Define a polygon')
268             self.setMinimumSize(700, 550)
269
270             self.polygon_points = polygon_points
271
272             # === Create the widgets
273
274             self._polygon_widget = DefinePolygonWidget(polygon_points=polygon_points)
275
276             button_confirm = QPushButton(self)
277             button_confirm.setText('Confirm')
278             button_confirm.clicked.connect(self._confirm_polygon)
279             button_confirm.setToolTip('Confirm this polygon<br><b>(Ctrl + Enter)</b>')
280             QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(button_confirm.click)
281
282             button_cancel = QPushButton(self)
283             button_cancel.setText('Cancel')
284             button_cancel.clicked.connect(self.reject)
285             button_cancel.setToolTip('Discard this polygon<br><b>(Escape)</b>')
286
287             button_reset = QPushButton(self)
288             button_reset.setText('Reset polygon')
289             button_reset.clicked.connect(self._polygon_widget.reset_polygon)
290             button_reset.setToolTip('Remove all points of the polygon<br><b>(Ctrl + R)</b>')
291             QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(button_reset.click)
292
293             # === Arrange the widgets
294
295             self.setContentsMargins(10, 10, 10, 10)
296
297             hlay_buttons = QHBoxLayout()
298             hlay_buttons.setSpacing(20)
299             hlay_buttons.addWidget(button_reset)
300             hlay_buttons.addItem(QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum))
301             hlay_buttons.addWidget(button_cancel)
302             hlay_buttons.addWidget(button_confirm)
303
```

```
304             vlay = QVBoxLayout()
305             vlay.setSpacing(20)
306             vlay.addWidget(self._polygon_widget)
307             vlay.addLayout(hlay_buttons)
308
309             self.setLayout(vlay)
310
311         @pyqtSlot()
312         def _confirm_polygon(self) -> None:
313             """Confirm the polygon that the user has defined."""
314             self.polygon_points = self._polygon_widget.points
315             self.accept()
```

A.15 gui/dialogs/settings.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides dialogs to edit settings within the app."""
8
9   from __future__ import annotations
10
11  import abc
12  from typing import Dict
13
14  from PyQt5 import QtWidgets
15  from PyQt5.QtGui import QIntValidator, QKeyEvent, QKeySequence
16  from PyQt5.QtWidgets import QCheckBox, QGroupBox, QHBoxLayout, QLayout, QShortcut, QSizePolicy, QSpacerItem,
    ↪  QVBoxLayout
17
18  from lintrans.gui.dialogs.misc import FixedSizeDialog
19  from lintrans.gui.settings import DisplaySettings
20
21
22  class SettingsDialog(FixedSizeDialog):
23      """An abstract superclass for other simple dialogs."""
24
25      def __init__(self, *args, resettable: bool, **kwargs):
26          """Create the widgets and layout of the dialog, passing ``*args`` and ``**kwargs`` to super."""
27          super().__init__(*args, **kwargs)
28
29          # === Create the widgets
30
31          self._button_confirm = QtWidgets.QPushButton(self)
32          self._button_confirm.setText('Confirm')
33          self._button_confirm.clicked.connect(self._confirm_settings)
34          self._button_confirm.setToolTip('Confirm these new settings<br><b>(Ctrl + Enter)</b>')
35          QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self._button_confirm.click)
36
37          self._button_cancel = QtWidgets.QPushButton(self)
38          self._button_cancel.setText('Cancel')
39          self._button_cancel.clicked.connect(self.reject)
40          self._button_cancel.setToolTip('Revert these settings<br><b>(Escape)</b>')
41
42          if resettable:
43              self._button_reset = QtWidgets.QPushButton(self)
44              self._button_reset.setText('Reset to defaults')
45              self._button_reset.clicked.connect(self._reset_settings)
46              self._button_reset.setToolTip('Reset these settings to their defaults<br><b>(Ctrl + R)</b>')
47              QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(self._button_reset.click)
48
49          # === Arrange the widgets
50
51          self.setContentsMargins(10, 10, 10, 10)
52
53          self._hlay_buttons = QHBoxLayout()
54          self._hlay_buttons.setSpacing(20)
```

```

        if resettable:
            self._hlay_buttons.addWidget(self._button_reset)

        self._hlay_buttons.addItem(QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum))
        self._hlay_buttons.addWidget(self._button_cancel)
        self._hlay_buttons.addWidget(self._button_confirm)

    def _setup_layout(self, options_layout: QLayout) -> None:
        """Set the layout of the settings widget.

        .. note:: This method must be called at the end of :meth:`__init__`
           in subclasses to setup the layout properly.
        """
        vlay_all = QVBoxLayout()
        vlay_all.setSpacing(20)
        vlay_all.addLayout(options_layout)
        vlay_all.addLayout(self._hlay_buttons)

        self.setLayout(vlay_all)

    @abc.abstractmethod
    def _load_settings(self) -> None:
        """Load the current settings into the widgets."""

    @abc.abstractmethod
    def _confirm_settings(self) -> None:
        """Confirm the settings chosen in the dialog."""

    def _reset_settings(self) -> None:
        """Reset the settings.

        .. note:: This method is empty but not abstract because not all subclasses will need to implement it.
        """


class DisplaySettingsDialog(SettingsDialog):
    """The dialog to allow the user to edit the display settings."""

    def __init__(self, *args, display_settings: DisplaySettings, **kwargs):
        """Create the widgets and layout of the dialog.

        :param DisplaySettings display_settings: The :class:`~lintrans.gui.settings.DisplaySettings` object to
    ↪  mutate
        """
        super().__init__(*args, resettable=True, **kwargs)

        self.display_settings = display_settings
        self.setWindowTitle('Change display settings')

        self._dict_checkboxes: Dict[str, QCheckBox] = {}

        # === Create the widgets

        # Basic stuff

        self._checkbox_draw_background_grid = QCheckBox(self)
        self._checkbox_draw_background_grid.setText('Draw &background grid')
        self._checkbox_draw_background_grid.setToolTip(
            'Draw the background grid (axes are always drawn)'
        )
        self._dict_checkboxes['b'] = self._checkbox_draw_background_grid

        self._checkbox_draw_transformed_grid = QCheckBox(self)
        self._checkbox_draw_transformed_grid.setText('Draw t&ransformed grid')
        self._checkbox_draw_transformed_grid.setToolTip(
            'Draw the transformed grid (vectors are handled separately)'
        )
        self._dict_checkboxes['r'] = self._checkbox_draw_transformed_grid

        self._checkbox_draw_basis_vectors = QCheckBox(self)
        self._checkbox_draw_basis_vectors.setText('Draw basis &vectors')
        self._checkbox_draw_basis_vectors.setToolTip(
```

```
            'Draw the transformed basis vectors'
        )
        self._checkbox_draw_basis_vectors.clicked.connect(self._update_gui)
        self._dict_checkboxes['v'] = self._checkbox_draw_basis_vectors

        self._checkbox_label_basis_vectors = QCheckBox(self)
        self._checkbox_label_basis_vectors.setText('Label the bas&is vectors')
        self._checkbox_label_basis_vectors.setToolTip(
            'Label the transformed i and j basis vectors'
        )
        self._dict_checkboxes['i'] = self._checkbox_label_basis_vectors

        # Animations

        self._checkbox_smoothen_determinant = QCheckBox(self)
        self._checkbox_smoothen_determinant.setText('&Smoothen determinant')
        self._checkbox_smoothen_determinant.setToolTip(
            'Smoothly animate the determinant transition during animation (if possible)'
        )
        self._dict_checkboxes['s'] = self._checkbox_smoothen_determinant

        self._checkbox_applicative_animation = QCheckBox(self)
        self._checkbox_applicative_animation.setText('&Applicative animation')
        self._checkbox_applicative_animation.setToolTip(
            'Animate the new transformation applied to the current one,\n'
            'rather than just that transformation on its own'
        )
        self._dict_checkboxes['a'] = self._checkbox_applicative_animation

        label_animation_time = QtWidgets.QLabel(self)
        label_animation_time.setText('Total animation length (ms)')
        label_animation_time.setToolTip(
            'How long it takes for an animation to complete'
        )

        self._lineedit_animation_time = QtWidgets.QLineEdit(self)
        self._lineedit_animation_time.setValidator(QIntValidator(1, 9999, self))
        self._lineedit_animation_time.textChanged.connect(self._update_gui)

        label_animation_pause_length = QtWidgets.QLabel(self)
        label_animation_pause_length.setText('Animation pause length (ms)')
        label_animation_pause_length.setToolTip(
            'How many milliseconds to pause for in comma-separated animations'
        )

        self._lineedit_animation_pause_length = QtWidgets.QLineEdit(self)
        self._lineedit_animation_pause_length.setValidator(QIntValidator(1, 999, self))

        # Matrix info

        self._checkbox_draw_determinant_parallelogram = QCheckBox(self)
        self._checkbox_draw_determinant_parallelogram.setText('Draw &determinant parallelogram')
        self._checkbox_draw_determinant_parallelogram.setToolTip(
            'Shade the parallelogram representing the determinant of the matrix'
        )
        self._checkbox_draw_determinant_parallelogram.clicked.connect(self._update_gui)
        self._dict_checkboxes['d'] = self._checkbox_draw_determinant_parallelogram

        self._checkbox_show_determinant_value = QCheckBox(self)
        self._checkbox_show_determinant_value.setText('Show de&terminant value')
        self._checkbox_show_determinant_value.setToolTip(
            'Show the value of the determinant inside the parallelogram'
        )
        self._dict_checkboxes['t'] = self._checkbox_show_determinant_value

        self._checkbox_draw_eigenvectors = QCheckBox(self)
        self._checkbox_draw_eigenvectors.setText('Draw &eigenvectors')
        self._checkbox_draw_eigenvectors.setToolTip('Draw the eigenvectors of the transformations')
        self._dict_checkboxes['e'] = self._checkbox_draw_eigenvectors

        self._checkbox_draw_eigenlines = QCheckBox(self)
        self._checkbox_draw_eigenlines.setText('Draw eigen&lines')
        self._checkbox_draw_eigenlines.setToolTip('Draw the eigenlines (invariant lines) of the transformations')
```

```
200             self._dict_checkboxes['l'] = self._checkbox_draw_eigenlines
201
202             # Polygon
203
204             self._checkbox_draw_untransformed_polygon = QCheckBox(self)
205             self._checkbox_draw_untransformed_polygon.setText('&Untransformed polygon')
206             self._checkbox_draw_untransformed_polygon.setToolTip('Draw the untransformed version of the polygon')
207             self._dict_checkboxes['u'] = self._checkbox_draw_untransformed_polygon
208
209             self._checkbox_draw_transformed_polygon = QCheckBox(self)
210             self._checkbox_draw_transformed_polygon.setText('Transformed &polygon')
211             self._checkbox_draw_transformed_polygon.setToolTip('Draw the transformed version of the polygon')
212             self._dict_checkboxes['p'] = self._checkbox_draw_transformed_polygon
213
214             # Input/output vectors
215
216             self._checkbox_draw_io_vectors = QCheckBox(self)
217             self._checkbox_draw_io_vectors.setText('Draw the vect&ors')
218             self._checkbox_draw_io_vectors.setToolTip('Draw the input and output vectors (only in the viewport)')
219             self._dict_checkboxes['o'] = self._checkbox_draw_io_vectors
220
221             # === Arrange the widgets in QGroupBoxes
222
223             # Basic stuff
224
225             vlay_groupbox_basic_stuff = QVBoxLayout()
226             vlay_groupbox_basic_stuff.setSpacing(20)
227             vlay_groupbox_basic_stuff.addWidget(self._checkbox_draw_background_grid)
228             vlay_groupbox_basic_stuff.addWidget(self._checkbox_draw_transformed_grid)
229             vlay_groupbox_basic_stuff.addWidget(self._checkbox_draw_basis_vectors)
230             vlay_groupbox_basic_stuff.addWidget(self._checkbox_label_basis_vectors)
231
232             groupbox_basic_stuff = QGroupBox('Basic stuff', self)
233             groupbox_basic_stuff.setLayout(vlay_groupbox_basic_stuff)
234
235             # Animations
236
237             hlay_animation_time = QHBoxLayout()
238             hlay_animation_time.addWidget(label_animation_time)
239             hlay_animation_time.addWidget(self._lineedit_animation_time)
240
241             hlay_animation_pause_length = QHBoxLayout()
242             hlay_animation_pause_length.addWidget(label_animation_pause_length)
243             hlay_animation_pause_length.addWidget(self._lineedit_animation_pause_length)
244
245             vlay_groupbox_animations = QVBoxLayout()
246             vlay_groupbox_animations.setSpacing(20)
247             vlay_groupbox_animations.addWidget(self._checkbox_smoothen_determinant)
248             vlay_groupbox_animations.addWidget(self._checkbox_applicative_animation)
249             vlay_groupbox_animations.addLayout(hlay_animation_time)
250             vlay_groupbox_animations.addLayout(hlay_animation_pause_length)
251
252             groupbox_animations = QGroupBox('Animations', self)
253             groupbox_animations.setLayout(vlay_groupbox_animations)
254
255             # Matrix info
256
257             vlay_groupbox_matrix_info = QVBoxLayout()
258             vlay_groupbox_matrix_info.setSpacing(20)
259             vlay_groupbox_matrix_info.addWidget(self._checkbox_draw_determinant_parallelogram)
260             vlay_groupbox_matrix_info.addWidget(self._checkbox_show_determinant_value)
261             vlay_groupbox_matrix_info.addWidget(self._checkbox_draw_eigenvectors)
262             vlay_groupbox_matrix_info.addWidget(self._checkbox_draw_eigenlines)
263
264             groupbox_matrix_info = QGroupBox('Matrix info', self)
265             groupbox_matrix_info.setLayout(vlay_groupbox_matrix_info)
266
267             # Polygon
268
269             vlay_groupbox_polygon = QVBoxLayout()
270             vlay_groupbox_polygon.setSpacing(20)
271             vlay_groupbox_polygon.addWidget(self._checkbox_draw_untransformed_polygon)
272             vlay_groupbox_polygon.addWidget(self._checkbox_draw_transformed_polygon)
```

```
273
274             groupbox_polygon = QGroupBox('Polygon', self)
275             groupbox_polygon.setLayout(vlay_groupbox_polygon)
276
277             # Input/output vectors
278
279             vlay_groupbox_io_vectors = QVBoxLayout()
280             vlay_groupbox_io_vectors.setSpacing(20)
281             vlay_groupbox_io_vectors.addWidget(self._checkbox_draw_io_vectors)
282
283             groupbox_io_vectors = QGroupBox('Input/output vectors', self)
284             groupbox_io_vectors.setLayout(vlay_groupbox_io_vectors)
285
286             # Now arrange the groupboxes
287             vlay_left = QVBoxLayout()
288             vlay_left.setSpacing(20)
289             vlay_left.addWidget(groupbox_basic_stuff)
290             vlay_left.addWidget(groupbox_animations)
291
292             vlay_right = QVBoxLayout()
293             vlay_right.setSpacing(20)
294             vlay_right.addWidget(groupbox_matrix_info)
295             vlay_right.addWidget(groupbox_polygon)
296             vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
297             vlay_right.addWidget(groupbox_io_vectors)
298
299             options_layout = QHBoxLayout()
300             options_layout.setSpacing(20)
301             options_layout.addLayout(vlay_left)
302             options_layout.addLayout(vlay_right)
303
304             self._setup_layout(options_layout)
305
306             # Finally, we load the current settings and update the GUI
307             self._load_settings()
308             self._update_gui()
309
310         def _load_settings(self) -> None:
311             """Load the current display settings into the widgets."""
312             # Basic stuff
313             self._checkbox_draw_background_grid.setChecked(self.display_settings.draw_background_grid)
314             self._checkbox_draw_transformed_grid.setChecked(self.display_settings.draw_transformed_grid)
315             self._checkbox_draw_basis_vectors.setChecked(self.display_settings.draw_basis_vectors)
316             self._checkbox_label_basis_vectors.setChecked(self.display_settings.label_basis_vectors)
317
318             # Animations
319             self._checkbox_smoothen_determinant.setChecked(self.display_settings.smoothen_determinant)
320             self._checkbox_applicative_animation.setChecked(self.display_settings.applicative_animation)
321             self._lineedit_animation_time.setText(str(self.display_settings.animation_time))
322             self._lineedit_animation_pause_length.setText(str(self.display_settings.animation_pause_length))
323
324             # Matrix info
325             self._checkbox_draw_determinant_parallelogram.setChecked(
                ↪  self.display_settings.draw_determinant_parallelogram)
326             self._checkbox_show_determinant_value.setChecked(self.display_settings.show_determinant_value)
327             self._checkbox_draw_eigenvectors.setChecked(self.display_settings.draw_eigenvectors)
328             self._checkbox_draw_eigenlines.setChecked(self.display_settings.draw_eigenlines)
329
330             # Polygon
331             self._checkbox_draw_untransformed_polygon.setChecked(self.display_settings.draw_untransformed_polygon)
332             self._checkbox_draw_transformed_polygon.setChecked(self.display_settings.draw_transformed_polygon)
333
334             # Input/output vectors
335             self._checkbox_draw_io_vectors.setChecked(self.display_settings.draw_io_vectors)
336
337         def _confirm_settings(self) -> None:
338             """Build a :class:`~lintrans.gui.settings.DisplaySettings` object and assign it."""
339             # Basic stuff
340             self.display_settings.draw_background_grid = self._checkbox_draw_background_grid.isChecked()
341             self.display_settings.draw_transformed_grid = self._checkbox_draw_transformed_grid.isChecked()
342             self.display_settings.draw_basis_vectors = self._checkbox_draw_basis_vectors.isChecked()
343             self.display_settings.label_basis_vectors = self._checkbox_label_basis_vectors.isChecked()
344
```

```
345             # Animations
346             self.display_settings.smoothen_determinant = self._checkbox_smoothen_determinant.isChecked()
347             self.display_settings.applicative_animation = self._checkbox_applicative_animation.isChecked()
348             self.display_settings.animation_time = int(self._lineedit_animation_time.text())
349             self.display_settings.animation_pause_length = int(self._lineedit_animation_pause_length.text())
350 
351             # Matrix info
352             self.display_settings.draw_determinant_parallelogram =
                ↪  self._checkbox_draw_determinant_parallelogram.isChecked()
353             self.display_settings.show_determinant_value = self._checkbox_show_determinant_value.isChecked()
354             self.display_settings.draw_eigenvectors = self._checkbox_draw_eigenvectors.isChecked()
355             self.display_settings.draw_eigenlines = self._checkbox_draw_eigenlines.isChecked()
356 
357             # Polygon
358             self.display_settings.draw_untransformed_polygon = self._checkbox_draw_untransformed_polygon.isChecked()
359             self.display_settings.draw_transformed_polygon = self._checkbox_draw_transformed_polygon.isChecked()
360 
361             # Input/output vectors
362             self.display_settings.draw_io_vectors = self._checkbox_draw_io_vectors.isChecked()
363 
364             self.accept()
365 
366         def _reset_settings(self) -> None:
367             """Reset the display settings to their defaults."""
368             self.display_settings = DisplaySettings()
369             self._load_settings()
370             self._update_gui()
371 
372         def _update_gui(self) -> None:
373             """Update the GUI according to other widgets in the GUI.
374 
375             For example, this method updates which checkboxes are enabled based on the values of other checkboxes.
376             """
377             self._checkbox_show_determinant_value.setEnabled(self._checkbox_draw_determinant_parallelogram.isChecked())
378             self._checkbox_label_basis_vectors.setEnabled(self._checkbox_draw_basis_vectors.isChecked())
379 
380             try:
381                 self._button_confirm.setEnabled(int(self._lineedit_animation_time.text()) >= 10)
382             except ValueError:
383                 self._button_confirm.setEnabled(False)
384 
385         def keyPressEvent(self, event: QKeyEvent) -> None:
386             """Handle a :class:`QKeyEvent` by manually activating toggling checkboxes.
387 
388             Qt handles these shortcuts automatically and allows the user to do ``Alt + Key``
389             to activate a simple shortcut defined with ``&``. However, I like to be able to
390             just hit ``Key`` and have the shortcut activate.
391             """
392             letter = event.text().lower()
393             key = event.key()
394 
395             if letter in self._dict_checkboxes:
396                 self._dict_checkboxes[letter].animateClick()
397 
398             # Return or keypad enter
399             elif key == 0x01000004 or key == 0x01000005:
400                 self._button_confirm.click()
401 
402             # Escape
403             elif key == 0x01000000:
404                 self._button_cancel.click()
405 
406             else:
407                 event.ignore()
```

A.16 gui/dialogs/define_new_matrix.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3 
```

```
    # This program is licensed under GNU GPLv3, available here:
    # <https://www.gnu.org/licenses/gpl-3.0.html>

    """This module provides an abstract :class:`DefineMatrixDialog` class and subclasses."""

    from __future__ import annotations

    import abc
    from typing import List, Tuple

    from numpy import array
    from PyQt5 import QtWidgets
    from PyQt5.QtCore import pyqtSlot
    from PyQt5.QtGui import QDoubleValidator, QKeySequence
    from PyQt5.QtWidgets import (QGridLayout, QHBoxLayout, QLabel, QLineEdit, QPushButton,
                                 QShortcut, QSizePolicy, QSpacerItem, QVBoxLayout)

    from lintrans.gui.dialogs.misc import FixedSizeDialog
    from lintrans.gui.plots import DefineMatrixVisuallyWidget
    from lintrans.gui.settings import DisplaySettings
    from lintrans.gui.validate import MatrixExpressionValidator
    from lintrans.matrices import MatrixWrapper
    from lintrans.matrices.utility import is_valid_float, round_float
    from lintrans.typing_ import MatrixType

    _ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'


    class DefineMatrixDialog(FixedSizeDialog):
        """An abstract superclass for definitions dialogs.

        .. warning:: This class should never be directly instantiated, only subclassed.
        """

        def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
            """Create the widgets and layout of the dialog.

            .. note:: ``*args`` and ``**kwargs`` are passed to the super constructor (:class:`QDialog`).

            :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
            """
            super().__init__(*args, **kwargs)

            self.matrix_wrapper = matrix_wrapper
            self.setWindowTitle('Define a matrix')

            # === Create the widgets

            self._button_confirm = QPushButton(self)
            self._button_confirm.setText('Confirm')
            self._button_confirm.setEnabled(False)
            self._button_confirm.clicked.connect(self._confirm_matrix)
            self._button_confirm.setToolTip('Confirm this as the new matrix<br><b>(Ctrl + Enter)</b>')
            QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self._button_confirm.click)

            button_cancel = QPushButton(self)
            button_cancel.setText('Cancel')
            button_cancel.clicked.connect(self.reject)
            button_cancel.setToolTip('Cancel this definition<br><b>(Escape)</b>')

            label_equals = QLabel(self)
            label_equals.setText('=')

            self._combobox_letter = QtWidgets.QComboBox(self)

            for letter in _ALPHABET_NO_I:
                self._combobox_letter.addItem(letter)

            self._combobox_letter.activated.connect(self._load_matrix)

            # === Arrange the widgets

            self.setContentsMargins(10, 10, 10, 10)
```

```

        self._hlay_buttons = QHBoxLayout()
        self._hlay_buttons.setSpacing(20)
        self._hlay_buttons.addItem(QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum))
        self._hlay_buttons.addWidget(button_cancel)
        self._hlay_buttons.addWidget(self._button_confirm)

        self._hlay_definition = QHBoxLayout()
        self._hlay_definition.setSpacing(20)
        self._hlay_definition.addWidget(self._combobox_letter)
        self._hlay_definition.addWidget(label_equals)

        # All subclasses have to manually add the hlay layouts to _vlay_all
        # This is because the subclasses add their own widgets and if we add
        # the layout here, then these new widgets won't be included
        self._vlay_all = QVBoxLayout()
        self._vlay_all.setSpacing(20)

        self.setLayout(self._vlay_all)

    @property
    def _selected_letter(self) -> str:
        """Return the letter currently selected in the combo box."""
        return str(self._combobox_letter.currentText())

    @abc.abstractmethod
    @pyqtSlot()
    def _update_confirm_button(self) -> None:
        """Enable the confirm button if it should be enabled, else, disable it."""

    @pyqtSlot(int)
    def _load_matrix(self, index: int) -> None:
        """Load the selected matrix into the dialog.

        This method is optionally able to be overridden. If it is not overridden,
        then no matrix is loaded when selecting a name.

        We have this method in the superclass so that we can define it as the slot
        for the :meth:`QComboBox.activated` signal in this constructor, rather than
        having to define that in the constructor of every subclass.
        """

    @abc.abstractmethod
    @pyqtSlot()
    def _confirm_matrix(self) -> None:
        """Confirm the inputted matrix and assign it.

        .. note:: When subclassing, this method should mutate ``self.matrix_wrapper`` and then call
    ↪  ``self.accept()``.
        """


class DefineVisuallyDialog(DefineMatrixDialog):
    """The dialog class that allows the user to define a matrix visually."""

    def __init__(
            self,
            *args,
            matrix_wrapper: MatrixWrapper,
            display_settings: DisplaySettings,
            polygon_points: List[Tuple[float, float]],
            **kwargs
    ):
        """Create the widgets and layout of the dialog.

        :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
        """
        super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)

        self.setMinimumSize(700, 550)

        # === Create the widgets

```

```
149             self._plot = DefineMatrixVisuallyWidget(
150                 self,
151                 display_settings=display_settings,
152                 polygon_points=polygon_points
153             )
154 
155             # === Arrange the widgets
156 
157             self._hlay_definition.addWidget(self._plot)
158             self._hlay_definition.setStretchFactor(self._plot, 1)
159 
160             self._vlay_all.addLayout(self._hlay_definition)
161             self._vlay_all.addLayout(self._hlay_buttons)
162 
163             # We load the default matrix A into the plot
164             self._load_matrix(0)
165 
166             # We also enable the confirm button, because any visually defined matrix is valid
167             self._button_confirm.setEnabled(True)
168 
169         @pyqtSlot()
170         def _update_confirm_button(self) -> None:
171             """Enable the confirm button.
172 
173             .. note::
174                The confirm button is always enabled in this dialog and this method is never actually used,
175                so it's got an empty body. It's only here because we need to implement the abstract method.
176             """
177 
178         @pyqtSlot(int)
179         def _load_matrix(self, index: int) -> None:
180             """Show the selected matrix on the plot. If the matrix is None, show the identity."""
181             matrix = self.matrix_wrapper[self._selected_letter]
182 
183             if matrix is None:
184                 matrix = self.matrix_wrapper['I']
185 
186             self._plot.plot_matrix(matrix)
187             self._plot.update()
188 
189         @pyqtSlot()
190         def _confirm_matrix(self) -> None:
191             """Confirm the matrix that's been defined visually."""
192             matrix: MatrixType = array([
193                 [self._plot.point_i[0], self._plot.point_j[0]],
194                 [self._plot.point_i[1], self._plot.point_j[1]]
195             ])
196 
197             self.matrix_wrapper[self._selected_letter] = matrix
198             self.accept()
199 
200 
201     class DefineNumericallyDialog(DefineMatrixDialog):
202         """The dialog class that allows the user to define a new matrix numerically."""
203 
204         def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
205             """Create the widgets and layout of the dialog.
206 
207             :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
208             """
209             super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
210 
211             # === Create the widgets
212 
213             # tl = top left, br = bottom right, etc.
214             self._element_tl = QLineEdit(self)
215             self._element_tl.textChanged.connect(self._update_confirm_button)
216             self._element_tl.setValidator(QDoubleValidator())
217 
218             self._element_tr = QLineEdit(self)
219             self._element_tr.textChanged.connect(self._update_confirm_button)
220             self._element_tr.setValidator(QDoubleValidator())
221 
```

```
222             self._element_bl = QLineEdit(self)
223             self._element_bl.textChanged.connect(self._update_confirm_button)
224             self._element_bl.setValidator(QDoubleValidator())
225
226             self._element_br = QLineEdit(self)
227             self._element_br.textChanged.connect(self._update_confirm_button)
228             self._element_br.setValidator(QDoubleValidator())
229
230             self._matrix_elements = (self._element_tl, self._element_tr, self._element_bl, self._element_br)
231
232             font_parens = self.font()
233             font_parens.setPointSize(int(font_parens.pointSize() * 5))
234             font_parens.setWeight(int(font_parens.weight() / 5))
235
236             label_paren_left = QLabel(self)
237             label_paren_left.setText('(')
238             label_paren_left.setFont(font_parens)
239
240             label_paren_right = QLabel(self)
241             label_paren_right.setText(')')
242             label_paren_right.setFont(font_parens)
243
244             # === Arrange the widgets
245
246             grid_matrix = QGridLayout()
247             grid_matrix.setSpacing(20)
248             grid_matrix.addWidget(label_paren_left, 0, 0, -1, 1)
249             grid_matrix.addWidget(self._element_tl, 0, 1)
250             grid_matrix.addWidget(self._element_tr, 0, 2)
251             grid_matrix.addWidget(self._element_bl, 1, 1)
252             grid_matrix.addWidget(self._element_br, 1, 2)
253             grid_matrix.addWidget(label_paren_right, 0, 3, -1, 1)
254
255             self._hlay_definition.addLayout(grid_matrix)
256
257             self._vlay_all.addLayout(self._hlay_definition)
258             self._vlay_all.addLayout(self._hlay_buttons)
259
260             # We load the default matrix A into the boxes
261             self._load_matrix(0)
262
263             self._element_tl.setFocus()
264
265         @pyqtSlot()
266         def _update_confirm_button(self) -> None:
267             """Enable the confirm button if there are valid floats in every box."""
268             for elem in self._matrix_elements:
269                 if not is_valid_float(elem.text()):
270                     # If they're not all numbers, then we can't confirm it
271                     self._button_confirm.setEnabled(False)
272                     return
273
274             # If we didn't find anything invalid
275             self._button_confirm.setEnabled(True)
276
277         @pyqtSlot(int)
278         def _load_matrix(self, index: int) -> None:
279             """If the selected matrix is defined, load its values into the boxes."""
280             matrix = self.matrix_wrapper[self._selected_letter]
281
282             if matrix is None:
283                 for elem in self._matrix_elements:
284                     elem.setText('')
285
286             else:
287                 self._element_tl.setText(round_float(matrix[0][0]))
288                 self._element_tr.setText(round_float(matrix[0][1]))
289                 self._element_bl.setText(round_float(matrix[1][0]))
290                 self._element_br.setText(round_float(matrix[1][1]))
291
292             self._update_confirm_button()
293
294         @pyqtSlot()
```

```
295         def _confirm_matrix(self) -> None:
296             """Confirm the matrix in the boxes and assign it to the name in the combo box."""
297             matrix: MatrixType = array([
298                 [float(self._element_tl.text()), float(self._element_tr.text())],
299                 [float(self._element_bl.text()), float(self._element_br.text())]
300             ])
301 
302             self.matrix_wrapper[self._selected_letter] = matrix
303             self.accept()
304 
305 
306     class DefineAsExpressionDialog(DefineMatrixDialog):
307         """The dialog class that allows the user to define a matrix as an expression of other matrices."""
308 
309         def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
310             """Create the widgets and layout of the dialog.
311 
312             :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
313             """
314             super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
315 
316             self.setMinimumWidth(450)
317 
318             # === Create the widgets
319 
320             self._lineedit_expression_box = QLineEdit(self)
321             self._lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
322             self._lineedit_expression_box.textChanged.connect(self._update_confirm_button)
323             self._lineedit_expression_box.setValidator(MatrixExpressionValidator())
324 
325             # === Arrange the widgets
326 
327             self._hlay_definition.addWidget(self._lineedit_expression_box)
328 
329             self._vlay_all.addLayout(self._hlay_definition)
330             self._vlay_all.addLayout(self._hlay_buttons)
331 
332             # Load the matrix if it's defined as an expression
333             self._load_matrix(0)
334 
335             self._lineedit_expression_box.setFocus()
336 
337         @pyqtSlot()
338         def _update_confirm_button(self) -> None:
339             """Enable the confirm button if the matrix expression is valid in the wrapper."""
340             text = self._lineedit_expression_box.text()
341             valid_expression = self.matrix_wrapper.is_valid_expression(text)
342 
343             self._button_confirm.setEnabled(
344                 valid_expression
345                 and self._selected_letter not in text
346                 and self._selected_letter not in self.matrix_wrapper.get_expression_dependencies(text)
347             )
348 
349         @pyqtSlot(int)
350         def _load_matrix(self, index: int) -> None:
351             """If the selected matrix is defined an expression, load that expression into the box."""
352             if (expr := self.matrix_wrapper.get_expression(self._selected_letter)) is not None:
353                 self._lineedit_expression_box.setText(expr)
354             else:
355                 self._lineedit_expression_box.setText('')
356 
357         @pyqtSlot()
358         def _confirm_matrix(self) -> None:
359             """Evaluate the matrix expression and assign its value to the name in the combo box."""
360             self.matrix_wrapper[self._selected_letter] = self._lineedit_expression_box.text()
361             self.accept()
```

A.17 gui/dialogs/__init__.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This package provides separate dialogs for the main GUI.

These dialogs are for defining new matrices in different ways and editing settings.
"""

from .define_new_matrix import (DefineAsExpressionDialog, DefineMatrixDialog,
                                DefineNumericallyDialog, DefineVisuallyDialog)
from .misc import AboutDialog, DefinePolygonDialog, FileSelectDialog, InfoPanelDialog
from .settings import DisplaySettingsDialog

__all__ = ['AboutDialog', 'DefineAsExpressionDialog', 'DefineMatrixDialog',
           'DefineNumericallyDialog', 'DefinePolygonDialog', 'DefineVisuallyDialog',
           'DisplaySettingsDialog', 'FileSelectDialog', 'InfoPanelDialog']
```

A.18 matrices/wrapper.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This module contains the main :class:`MatrixWrapper` class and a function to create a matrix from an angle."""

from __future__ import annotations

import re
from copy import copy
from functools import reduce
from operator import add, matmul
from typing import Any, Dict, List, Optional, Set, Tuple, Union

import numpy as np

from lintrans.typing_ import is_matrix_type, MatrixType
from .parse import get_matrix_identifiers, parse_matrix_expression, validate_matrix_expression
from .utility import create_rotation_matrix


class MatrixWrapper:
    """A wrapper class to hold all possible matrices and allow access to them.

    .. note::
       When defining a custom matrix, its name must be a capital letter and cannot be ``I``.

    The contained matrices can be accessed and assigned to using square bracket notation.

    :Example:

    >>> wrapper = MatrixWrapper()
    >>> wrapper['I']
    array([[1., 0.],
           [0., 1.]])
    >>> wrapper['M']  # Returns None
    >>> wrapper['M'] = np.array([[1, 2], [3, 4]])
    >>> wrapper['M']
    array([[1., 2.],
           [3., 4.]])
    """

    def __init__(self):
```

```python
        """Initialize a :class:`MatrixWrapper` object with a dictionary of matrices which can be accessed."""
        self._matrices: Dict[str, Optional[Union[MatrixType, str]]] = {
            'A': None, 'B': None, 'C': None, 'D': None,
            'E': None, 'F': None, 'G': None, 'H': None,
            'I': np.eye(2),  # I is always defined as the identity matrix
            'J': None, 'K': None, 'L': None, 'M': None,
            'N': None, 'O': None, 'P': None, 'Q': None,
            'R': None, 'S': None, 'T': None, 'U': None,
            'V': None, 'W': None, 'X': None, 'Y': None,
            'Z': None
        }

    def __repr__(self) -> str:
        """Return a nice string repr of the :class:`MatrixWrapper` for debugging."""
        defined_matrices = ''.join([k for k, v in self._matrices.items() if v is not None])
        return f'<{self.__class__.__module__}.{self.__class__.__name__} object with ' \
               f"{len(defined_matrices)} defined matrices: '{defined_matrices}'>"

    def __eq__(self, other: Any) -> bool:
        """Check for equality in wrappers by comparing dictionaries.

        :param Any other: The object to compare this wrapper to
        """
        if not isinstance(other, self.__class__):
            return NotImplemented

        # We loop over every matrix and check if every value is equal in each
        for name in self._matrices:
            s_matrix = self[name]
            o_matrix = other[name]

            if s_matrix is None and o_matrix is None:
                continue

            elif (s_matrix is None and o_matrix is not None) or \
                 (s_matrix is not None and o_matrix is None):
                return False

            # This is mainly to satisfy mypy, because we know these must be matrices
            elif not is_matrix_type(s_matrix) or not is_matrix_type(o_matrix):
                return False

            # Now we know they're both NumPy arrays
            elif np.array_equal(s_matrix, o_matrix):
                continue

            else:
                return False

        return True

    def __hash__(self) -> int:
        """Return the hash of the matrices dictionary."""
        return hash(self._matrices)

    def __getitem__(self, name: str) -> Optional[MatrixType]:
        """Get the matrix with the given name.

        If it is a simple name, it will just be fetched from the dictionary. If the name is ``rot(x)``, with
        a given angle in degrees, then we return a new matrix representing a rotation by that angle.

        .. note::
           If the named matrix is defined as an expression, then this method will return its evaluation.
           If you want the expression itself, use :meth:`get_expression`.

        :param str name: The name of the matrix to get
        :returns Optional[MatrixType]: The value of the matrix (could be None)

        :raises NameError: If there is no matrix with the given name
        """
        # Return a new rotation matrix
        if (match := re.match(r'^rot\((-?\d*\.?\d*)\)$', name)) is not None:
            return create_rotation_matrix(float(match.group(1)))
```

```
119
120             if name not in self._matrices:
121                 if validate_matrix_expression(name):
122                     return self.evaluate_expression(name)
123
124                 raise NameError(f'Unrecognised matrix name "{name}"')
125
126             # We copy the matrix before we return it so the user can't accidentally mutate the matrix
127             matrix = copy(self._matrices[name])
128
129             if isinstance(matrix, str):
130                 return self.evaluate_expression(matrix)
131
132             return matrix
133
134         def __setitem__(self, name: str, new_matrix: Optional[Union[MatrixType, str]]) -> None:
135             """Set the value of matrix ``name`` with the new_matrix.
136
137             The new matrix may be a simple 2x2 NumPy array, or it could be a string, representing an
138             expression in terms of other, previously defined matrices.
139
140             :param str name: The name of the matrix to set the value of
141             :param Optional[Union[MatrixType, str]] new_matrix: The value of the new matrix (could be None)
142
143             :raises NameError: If the name isn't a legal matrix name
144             :raises TypeError: If the matrix isn't a valid 2x2 NumPy array or expression in terms of other defined
    ↪  matrices
145             :raises ValueError: If you attempt to define a matrix in terms of itself
146             """
147             if not (name in self._matrices and name != 'I'):
148                 raise NameError('Matrix name is illegal')
149
150             if new_matrix is None:
151                 self._matrices[name] = None
152                 return
153
154             if isinstance(new_matrix, str):
155                 if self.is_valid_expression(new_matrix):
156                     if name not in new_matrix and \
157                             name not in self.get_expression_dependencies(new_matrix):
158                         self._matrices[name] = new_matrix
159                         return
160                     else:
161                         raise ValueError('Cannot define a matrix recursively')
162
163             if not is_matrix_type(new_matrix):
164                 raise TypeError('Matrix must be a 2x2 NumPy array')
165
166             # All matrices must have float entries
167             a = float(new_matrix[0][0])
168             b = float(new_matrix[0][1])
169             c = float(new_matrix[1][0])
170             d = float(new_matrix[1][1])
171
172             self._matrices[name] = np.array([[a, b], [c, d]])
173
174         def get_matrix_dependencies(self, matrix_name: str) -> Set[str]:
175             """Return all the matrices (as identifiers) that the given matrix (indirectly) depends on.
176
177             If A depends on nothing, B directly depends on A, and C directly depends on B,
178             then we say C depends on B `and` A.
179             """
180             expression = self.get_expression(matrix_name)
181             if expression is None:
182                 return set()
183
184             s = set()
185             identifiers = get_matrix_identifiers(expression)
186             for identifier in identifiers:
187                 s.add(identifier)
188                 s.update(self.get_matrix_dependencies(identifier))
189
190             return s
```

```
191
192         def get_expression_dependencies(self, expression: str) -> Set[str]:
193             """Return all the matrices that the given expression depends on.
194
195             This method just calls :meth:`get_matrix_dependencies` on each matrix
196             identifier in the expression. See that method for details.
197
198             If an expression contains a matrix that has no dependencies, then the
199             expression is `not` considered to depend on that matrix. But it `is`
200             considered to depend on any matrix that has its own dependencies.
201             """
202             s = set()
203             for iden in get_matrix_identifiers(expression):
204                 s.update(self.get_matrix_dependencies(iden))
205             return s
206
207         def get_expression(self, name: str) -> Optional[str]:
208             """If the named matrix is defined as an expression, return that expression, else return None.
209
210             :param str name: The name of the matrix
211             :returns Optional[str]: The expression that the matrix is defined as, or None
212
213             :raises NameError: If the name is invalid
214             """
215             if name not in self._matrices:
216                 raise NameError('Matrix must have a legal name')
217
218             matrix = self._matrices[name]
219             if isinstance(matrix, str):
220                 return matrix
221
222             return None
223
224         def is_valid_expression(self, expression: str) -> bool:
225             """Check if the given expression is valid, using the context of the wrapper.
226
227             This method calls :func:`lintrans.matrices.parse.validate_matrix_expression`, but also
228             ensures that all the matrices in the expression are defined in the wrapper.
229
230             :param str expression: The expression to validate
231             :returns bool: Whether the expression is valid in this wrapper
232
233             :raises LinAlgError: If a matrix is defined in terms of the inverse of a singular matrix
234             """
235             # Get rid of the transposes to check all capital letters
236             new_expression = expression.replace('^T', '').replace('^{T}', '')
237
238             # Make sure all the referenced matrices are defined
239             for matrix in [x for x in new_expression if re.match('[A-Z]', x)]:
240                 if self[matrix] is None:
241                     return False
242
243                 if (expr := self.get_expression(matrix)) is not None:
244                     if not self.is_valid_expression(expr):
245                         return False
246
247             return validate_matrix_expression(expression)
248
249         def evaluate_expression(self, expression: str) -> MatrixType:
250             """Evaluate a given expression and return the matrix evaluation.
251
252             :param str expression: The expression to be parsed
253             :returns MatrixType: The matrix result of the expression
254
255             :raises ValueError: If the expression is invalid
256             """
257             if not self.is_valid_expression(expression):
258                 raise ValueError('The expression is invalid')
259
260             parsed_result = parse_matrix_expression(expression)
261             final_groups: List[List[MatrixType]] = []
262
263             for group in parsed_result:
```

```
            f_group: List[MatrixType] = []

            for multiplier, identifier, index in group:
                if index == 'T':
                    m = self[identifier]

                    # This assertion is just so mypy doesn't complain
                    # We know this won't be None, because we know that this matrix is defined in this wrapper
                    assert m is not None
                    matrix_value = m.T

                else:
                    # Again, this assertion is just for mypy
                    # We know this will be a matrix, but since upgrading from NumPy 1.21 to 1.23
                    # (to fix a bug with GH Actions on Windows), mypy complains about matrix_power()
                    base_matrix = self[identifier]
                    assert is_matrix_type(base_matrix)

                    matrix_value = np.linalg.matrix_power(base_matrix, 1 if index == '' else int(index))

                matrix_value *= 1 if multiplier == '' else float(multiplier)
                f_group.append(matrix_value)

            final_groups.append(f_group)

        return reduce(add, [reduce(matmul, group) for group in final_groups])

    def get_defined_matrices(self) -> List[Tuple[str, Union[MatrixType, str]]]:
        """Return a list of tuples containing the name and value of all defined matrices in the wrapper.

        :returns: A list of tuples where the first element is the name, and the second element is the value
        :rtype: List[Tuple[str, Union[MatrixType, str]]]
        """
        matrices = []

        for name, value in self._matrices.items():
            if value is not None:
                matrices.append((name, value))

        return matrices
```

A.19 matrices/utility.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This module provides simple utility methods for matrix and vector manipulation."""

from __future__ import annotations

import math
from typing import Tuple

import numpy as np

from lintrans.typing_ import MatrixType


def polar_coords(x: float, y: float, *, degrees: bool = False) -> Tuple[float, float]:
    r"""Return the polar coordinates of a given (x, y) Cartesian coordinate.

    .. note:: We're returning the angle in the range :math:`[0, 2\pi)`
    """
    radius = math.hypot(x, y)

    # PyCharm complains about np.angle taking a complex argument even though that's what it's designed for
    # noinspection PyTypeChecker
```

```python
    angle = float(np.angle(x + y * 1j, degrees))

    if angle < 0:
        angle += 2 * np.pi

    return radius, angle


def rect_coords(radius: float, angle: float, *, degrees: bool = False) -> Tuple[float, float]:
    """Return the rectilinear coordinates of a given polar coordinate."""
    if degrees:
        angle = np.radians(angle)

    return radius * np.cos(angle), radius * np.sin(angle)


def rotate_coord(x: float, y: float, angle: float, *, degrees: bool = False) -> Tuple[float, float]:
    """Rotate a rectilinear coordinate by the given angle."""
    if degrees:
        angle = np.radians(angle)

    r, theta = polar_coords(x, y, degrees=degrees)
    theta = (theta + angle) % (2 * np.pi)

    return rect_coords(r, theta, degrees=degrees)


def create_rotation_matrix(angle: float, *, degrees: bool = True) -> MatrixType:
    """Create a matrix representing a rotation (anticlockwise) by the given angle.

    :Example:

    >>> create_rotation_matrix(30)
    array([[ 0.8660254, -0.5      ],
           [ 0.5      ,  0.8660254]])
    >>> create_rotation_matrix(45)
    array([[ 0.70710678, -0.70710678],
           [ 0.70710678,  0.70710678]])
    >>> create_rotation_matrix(np.pi / 3, degrees=False)
    array([[ 0.5      , -0.8660254],
           [ 0.8660254,  0.5      ]])

    :param float angle: The angle to rotate anticlockwise by
    :param bool degrees: Whether to interpret the angle as degrees (True) or radians (False)
    :returns MatrixType: The resultant matrix
    """
    rad = np.deg2rad(angle % 360) if degrees else angle % (2 * np.pi)
    return np.array([
        [np.cos(rad), -1 * np.sin(rad)],
        [np.sin(rad), np.cos(rad)]
    ])


def is_valid_float(string: str) -> bool:
    """Check if the string is a valid float (or anything that can be cast to a float, such as an int).

    This function simply checks that ``float(string)`` doesn't raise an error.

    .. note:: An empty string is not a valid float, so will return False.

    :param str string: The string to check
    :returns bool: Whether the string is a valid float
    """
    try:
        float(string)
        return True
    except ValueError:
        return False


def round_float(num: float, precision: int = 5) -> str:
    """Round a floating point number to a given number of decimal places for pretty printing.

```

```
        :param float num: The number to round
        :param int precision: The number of decimal places to round to
        :returns str: The rounded number for pretty printing
        """
        # Round to ``precision`` number of decimal places
        string = str(round(num, precision))

        # Cut off the potential final zero
        if string.endswith('.0'):
            return string[:-2]

        elif 'e' in string:  # Scientific notation
            split = string.split('e')
            # The leading 0 only happens when the exponent is negative, so we know there'll be a minus sign
            return split[0] + 'e-' + split[1][1:].lstrip('0')

        else:
            return string
```

A.20 `matrices/parse.py`

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This module provides functions to parse and validate matrix expressions."""

from __future__ import annotations

import re
from dataclasses import dataclass
from typing import List, Pattern, Tuple, Set

from lintrans.typing_ import MatrixParseList

_ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'

NAIVE_CHARACTER_CLASS = r'[-+\sA-Z0-9.rot()^{}]'
"""This is a RegEx character class that just holds all the valid characters for an expression.

See :func:`validate_matrix_expression` to actually validate matrix expressions.
"""


class MatrixParseError(Exception):
    """A simple exception to be raised when an error is found when parsing."""


def compile_naive_expression_pattern() -> Pattern[str]:
    """Compile the single RegEx pattern that will match a valid matrix expression."""
    digit_no_zero = '[123456789]'
    digits = '\\d+'
    integer_no_zero = digit_no_zero + '(' + digits + ')?'
    real_number = f'({integer_no_zero}(\\.{digits})?|0\\.{digits})'

    index_content = f'(-?{integer_no_zero}|T)'
    index = f'(\\^{{{index_content}}}|\\^{index_content})'
    matrix_identifier = f'([A-Z]|rot\\(-?{real_number}\\)|\\({NAIVE_CHARACTER_CLASS}+\\))'
    matrix = '(' + real_number + '?' + matrix_identifier + index + '?)'
    expression = f'^-?{matrix}+((\\+-?|-){matrix}+)*$'

    return re.compile(expression)


# This is an expensive pattern to compile, so we compile it when this module is initialized
_naive_expression_pattern = compile_naive_expression_pattern()
```

```
50      def find_sub_expressions(expression: str) -> List[str]:
51          """Find all the sub-expressions in the given expression.
52
53          This function only goes one level deep, so may return strings like ``'A(BC)D'``.
54
55          :raises MatrixParseError: If there are unbalanced parentheses
56          """
57          sub_expressions: List[str] = []
58          string = ''
59          paren_depth = 0
60          pointer = 0
61
62          while True:
63              char = expression[pointer]
64
65              if char == '(' and expression[pointer - 3:pointer] != 'rot':
66                  paren_depth += 1
67
68                  # This is a bit of a manual bodge, but it eliminates extraneous parens
69                  if paren_depth == 1:
70                      pointer += 1
71                      continue
72
73              elif char == ')' and re.match(f'{NAIVE_CHARACTER_CLASS}*?rot\\([-\\d.]+$', expression[:pointer]) is None:
74                  paren_depth -= 1
75
76              if paren_depth > 0:
77                  string += char
78
79              if paren_depth == 0 and string:
80                  sub_expressions.append(string)
81                  string = ''
82
83              pointer += 1
84
85              if pointer >= len(expression):
86                  break
87
88          if paren_depth != 0:
89              raise MatrixParseError('Unbalanced parentheses in expression')
90
91          return sub_expressions
92
93
94      def validate_matrix_expression(expression: str) -> bool:
95          """Validate the given matrix expression.
96
97          This function simply checks the expression against the BNF schema documented in
98          :ref:`expression-syntax-docs`. It is not aware of which matrices are actually defined
99          in a wrapper. For an aware version of this function, use the
100         :meth:`~lintrans.matrices.wrapper.MatrixWrapper.is_valid_expression` method on
101         :class:`~lintrans.matrices.wrapper.MatrixWrapper`.
102
103         :param str expression: The expression to be validated
104         :returns bool: Whether the expression is valid according to the schema
105         """
106         # Remove all whitespace
107         expression = re.sub(r'\s', '', expression)
108         match = _naive_expression_pattern.match(expression)
109
110         if match is None:
111             return False
112
113         if re.search(r'\^-?\d*\.\d+', expression) is not None:
114             return False
115
116         # Check that the whole expression was matched against
117         if expression != match.group(0):
118             return False
119
120         try:
121             sub_expressions = find_sub_expressions(expression)
122         except MatrixParseError:
```

```
123             return False
124
125         if len(sub_expressions) == 0:
126             return True
127
128         return all(validate_matrix_expression(m) for m in sub_expressions)
129
130
131     @dataclass
132     class MatrixToken:
133         """A simple dataclass to hold information about a matrix token being parsed."""
134
135         multiplier: str = ''
136         identifier: str = ''
137         exponent: str = ''
138
139         @property
140         def tuple(self) -> Tuple[str, str, str]:
141             """Create a tuple of the token for parsing."""
142             return self.multiplier, self.identifier, self.exponent
143
144
145     class ExpressionParser:
146         """A class to hold state during parsing.
147
148         Most of the methods in this class are class-internal and should not be used from outside.
149
150         This class should be used like this:
151
152         >>> ExpressionParser('3A^-1B').parse()
153         [[('3', 'A', '-1'), ('', 'B', '')]]
154         >>> ExpressionParser('4(M^TA^2)^-2').parse()
155         [[('4', 'M^{T}A^{2}', '-2')]]
156         """
157
158         def __init__(self, expression: str):
159             """Create an instance of the parser with the given expression and initialise variables to use during
                ↪  parsing."""
160             # Remove all whitespace
161             expression = re.sub(r'\s', '', expression)
162
163             # Check if it's valid
164             if not validate_matrix_expression(expression):
165                 raise MatrixParseError('Invalid expression')
166
167             # Wrap all exponents and transposition powers with {}
168             expression = re.sub(r'(?<=\^)(-?\d+|T)(?=[^}]|$)', r'{\g<0>}', expression)
169
170             # Remove any standalone minuses
171             expression = re.sub(r'-(?=[A-Z])', '-1', expression)
172
173             # Replace subtractions with additions
174             expression = re.sub(r'-(?=\d+\.?\d*([A-Z]|rot))', '+-', expression)
175
176             # Get rid of a potential leading + introduced by the last step
177             expression = re.sub(r'^\+', '', expression)
178
179             self._expression = expression
180             self._pointer: int = 0
181
182             self._current_token = MatrixToken()
183             self._current_group: List[Tuple[str, str, str]] = []
184
185             self._final_list: MatrixParseList = []
186
187         def __repr__(self) -> str:
188             """Return a simple repr containing the expression."""
189             return f'{self.__class__.__module__}.{self.__class__.__name__}("{self._expression}")'
190
191         @property
192         def _char(self) -> str:
193             """Return the character pointed to by the pointer."""
194             return self._expression[self._pointer]
```

```
195
196         def parse(self) -> MatrixParseList:
197             """Fully parse the instance's matrix expression and return the :attr:`~lintrans.typing_.MatrixParseList`.
198
199             This method uses all the private methods of this class to parse the
200             expression in parts. All private methods mutate the instance variables.
201
202             :returns: The parsed expression
203             :rtype: :attr:`~lintrans.typing_.MatrixParseList`
204             """
205             self._parse_multiplication_group()
206
207             while self._pointer < len(self._expression):
208                 if self._expression[self._pointer] != '+':
209                     raise MatrixParseError('Expected "+" between multiplication groups')
210
211                 self._pointer += 1
212                 self._parse_multiplication_group()
213
214             return self._final_list
215
216         def _parse_multiplication_group(self) -> None:
217             """Parse a group of matrices to be multiplied together.
218
219             This method just parses matrices until we get to a ``+``.
220             """
221             # This loop continues to parse matrices until we fail to do so
222             while self._parse_matrix():
223                 # Once we get to the end of the multiplication group, we add it the final list and reset the group list
224                 if self._pointer >= len(self._expression) or self._char == '+':
225                     self._final_list.append(self._current_group)
226                     self._current_group = []
227                     self._pointer += 1
228
229         def _parse_matrix(self) -> bool:
230             """Parse a full matrix using :meth:`_parse_matrix_part`.
231
232             This method will parse an optional multiplier, an identifier, and an optional exponent. If we
233             do this successfully, we return True. If we fail to parse a matrix (maybe we've reached the
234             end of the current multiplication group and the next char is ``+``), then we return False.
235
236             :returns bool: Success or failure
237             """
238             self._current_token = MatrixToken()
239
240             while self._parse_matrix_part():
241                 pass  # The actual execution is taken care of in the loop condition
242
243             if self._current_token.identifier == '':
244                 return False
245
246             self._current_group.append(self._current_token.tuple)
247             return True
248
249         def _parse_matrix_part(self) -> bool:
250             """Parse part of a matrix (multiplier, identifier, or exponent).
251
252             Which part of the matrix we parse is dependent on the current value of the pointer and the expression.
253             This method will parse whichever part of matrix token that it can. If it can't parse a part of a matrix,
254             or it's reached the next matrix, then we just return False. If we succeeded to parse a matrix part, then
255             we return True.
256
257             :returns bool: Success or failure
258             :raises MatrixParseError: If we fail to parse this part of the matrix
259             """
260             if self._pointer >= len(self._expression):
261                 return False
262
263             if self._char.isdigit() or self._char == '-':
264                 if self._current_token.multiplier != '' \
265                         or (self._current_token.multiplier == '' and self._current_token.identifier != ''):
266                     return False
267
```

```
268                 self._parse_multiplier()
269
270             elif self._char.isalpha() and self._char.isupper():
271                 if self._current_token.identifier != '':
272                     return False
273
274                 self._current_token.identifier = self._char
275                 self._pointer += 1
276
277             elif self._char == 'r':
278                 if self._current_token.identifier != '':
279                     return False
280
281                 self._parse_rot_identifier()
282
283             elif self._char == '(':
284                 if self._current_token.identifier != '':
285                     return False
286
287                 self._parse_sub_expression()
288
289             elif self._char == '^':
290                 if self._current_token.exponent != '':
291                     return False
292
293                 self._parse_exponent()
294
295             elif self._char == '+':
296                 return False
297
298             else:
299                 raise MatrixParseError(f'Unrecognised character "{self._char}" in matrix expression')
300
301             return True
302
303         def _parse_multiplier(self) -> None:
304             """Parse a multiplier from the expression and pointer.
305
306             This method just parses a numerical multiplier, which can include
307             zero or one ``.`` character and optionally a ``-`` at the start.
308
309             :raises MatrixParseError: If we fail to parse this part of the matrix
310             """
311             multiplier = ''
312
313             while self._char.isdigit() or self._char in ('.', '-'):
314                 multiplier += self._char
315                 self._pointer += 1
316
317             try:
318                 float(multiplier)
319             except ValueError as e:
320                 raise MatrixParseError(f'Invalid multiplier "{multiplier}"') from e
321
322             self._current_token.multiplier = multiplier
323
324         def _parse_rot_identifier(self) -> None:
325             """Parse a ``rot()``-style identifier from the expression and pointer.
326
327             This method will just parse something like ``rot(12.5)``. The angle number must be a real number.
328
329             :raises MatrixParseError: If we fail to parse this part of the matrix
330             """
331             if match := re.match(r'rot\(([\d.-]+)\)', self._expression[self._pointer:]):
332                 # Ensure that the number in brackets is a valid float
333                 try:
334                     float(match.group(1))
335                 except ValueError as e:
336                     raise MatrixParseError(f'Invalid angle number "{match.group(1)}" in rot-identifier') from e
337
338                 self._current_token.identifier = match.group(0)
339                 self._pointer += len(match.group(0))
340             else:
```

```
                raise MatrixParseError(
                    f'Invalid rot-identifier "{self._expression[self._pointer : self._pointer + 15]}..."'
                )

    def _parse_sub_expression(self) -> None:
        """Parse a parenthesized sub-expression as the identifier.

        This method will also validate the expression in the parentheses.

        :raises MatrixParseError: If we fail to parse this part of the matrix
        """
        if self._char != '(':
            raise MatrixParseError('Sub-expression must start with "("')

        self._pointer += 1
        paren_depth = 1
        identifier = ''

        while paren_depth > 0:
            if self._char == '(':
                paren_depth += 1
            elif self._char == ')':
                paren_depth -= 1

            if paren_depth == 0:
                self._pointer += 1
                break

            identifier += self._char
            self._pointer += 1

        if not validate_matrix_expression(identifier):
            raise MatrixParseError(f'Invalid sub-expression identifier "{identifier}"')

        self._current_token.identifier = identifier

    def _parse_exponent(self) -> None:
        """Parse a matrix exponent from the expression and pointer.

        The exponent must be an integer or ``T`` for transpose.

        :raises MatrixParseError: If we fail to parse this part of the token
        """
        if match := re.match(r'\^\{(-?\d+|T)\}', self._expression[self._pointer:]):
            exponent = match.group(1)

            try:
                if exponent != 'T':
                    int(exponent)
            except ValueError as e:
                raise MatrixParseError(f'Invalid exponent "{match.group(1)}"') from e

            self._current_token.exponent = exponent
            self._pointer += len(match.group(0))
        else:
            raise MatrixParseError(
                f'Invalid exponent "{self._expression[self._pointer : self._pointer + 10]}..."'
            )


def parse_matrix_expression(expression: str) -> MatrixParseList:
    """Parse the matrix expression and return a :attr:`~lintrans.typing_.MatrixParseList`.

    :Example:

    >>> parse_matrix_expression('A')
    [[('', 'A', '')]]
    >>> parse_matrix_expression('-3M^2')
    [[('-3', 'M', '2')]]
    >>> parse_matrix_expression('1.2rot(12)^{3}2B^T')
    [[('1.2', 'rot(12)', '3'), ('2', 'B', 'T')]]
    >>> parse_matrix_expression('A^2 + 3B')
    [[('', 'A', '2')], [('3', 'B', '')]]
```

```
        >>> parse_matrix_expression('-3A^{-1}3B^T - 45M^2')
        [[('-3', 'A', '-1'), ('3', 'B', 'T')], [('-45', 'M', '2')]]
        >>> parse_matrix_expression('5.3A^{4} 2.6B^{-2} + 4.6D^T 8.9E^{-1}')
        [[('5.3', 'A', '4'), ('2.6', 'B', '-2')], [('4.6', 'D', 'T'), ('8.9', 'E', '-1')]]
        >>> parse_matrix_expression('2(A+B^TC)^2D')
        [[('2', 'A+B^{T}C', '2'), ('', 'D', '')]]

        :param str expression: The expression to be parsed
        :returns: A list of parsed components
        :rtype: :attr:`~lintrans.typing_.MatrixParseList`
        """
        return ExpressionParser(expression).parse()


    def get_matrix_identifiers(expression: str) -> Set[str]:
        """Return all the matrix identifiers used in the given expression.

        This method works recursively with sub-expressions.
        """
        s = set()
        top_level = [id for sublist in parse_matrix_expression(expression) for _, id, _ in sublist]

        for body in top_level:
            if body in _ALPHABET:
                s.add(body)

            elif re.match(r'rot\(\d+(\.\d+)?\)', body):
                continue

            else:
                s.update(get_matrix_identifiers(body))

        return s
```

A.21 matrices/__init__.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This package supplies classes and functions to parse, evaluate, and wrap matrices."""

from . import parse, utility
from .utility import create_rotation_matrix
from .wrapper import MatrixWrapper

__all__ = ['create_rotation_matrix', 'MatrixWrapper', 'parse', 'utility']
```

A.22 typing_/__init__.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""This package supplies type aliases for linear algebra and transformations.

.. note::
   This package is called ``typing_`` and not ``typing`` to avoid name collisions with the
   builtin :mod:`typing`. I don't quite know how this collision occurs, but renaming
   this module fixed the problem.
"""

from __future__ import annotations

```

```
17      from sys import version_info
18      from typing import Any, List, Tuple
19
20      from numpy import ndarray
21      from nptyping import NDArray, Float
22
23      if version_info >= (3, 10):
24          from typing import TypeAlias, TypeGuard
25
26      __all__ = ['is_matrix_type', 'MatrixType', 'MatrixParseList', 'VectorType']
27
28      MatrixType: TypeAlias = 'NDArray[(2, 2), Float]'
29      """This type represents a 2x2 matrix as a NumPy array."""
30
31      VectorType: TypeAlias = 'NDArray[(2,), Float]'
32      """This type represents a 2D vector as a NumPy array, for use with :attr:`MatrixType`."""
33
34      MatrixParseList: TypeAlias = List[List[Tuple[str, str, str]]]
35      """This is a list containing lists of tuples. Each tuple represents a matrix and is ``(multiplier,
36      matrix_identifier, index)`` where all of them are strings. These matrix-representing tuples are
37      contained in lists which represent multiplication groups. Every matrix in the group should be
38      multiplied together, in order. These multiplication group lists are contained by a top level list,
39      which is this type. Once these multiplication group lists have been evaluated, they should be summed.
40
41      In the tuples, the multiplier is a string representing a real number, the matrix identifier
42      is a capital letter or ``rot(x)`` where x is a real number angle, and the index is a string
43      representing an integer, or it's the letter ``T`` for transpose.
44      """
45
46
47      def is_matrix_type(matrix: Any) -> TypeGuard[MatrixType]:
48          """Check if the given value is a valid matrix type.
49
50          .. note::
51             This function is a TypeGuard, meaning if it returns True, then the
52             passed value must be a :attr:`MatrixType`.
53          """
54          return isinstance(matrix, ndarray) and matrix.shape == (2, 2)
```

B Testing code

B.1 `conftest.py`

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""A simple ``conftest.py`` containing some re-usable fixtures and functions."""

import os
from typing import List, Type, TypeVar

import numpy as np
import pytest
from _pytest.config import Config
from _pytest.python import Function
from PyQt5.QtWidgets import QApplication, QWidget
from pytestqt.qtbot import QtBot

from lintrans.gui.main_window import LintransMainWindow
from lintrans.matrices import MatrixWrapper


T = TypeVar('T', bound=QWidget)


def pytest_collection_modifyitems(config: Config, items: List[Function]) -> None:
    """Modify the collected tests so that we only run the GUI tests on Linux (because they need an X server).

    This function is called automatically during the pytest startup. See
    https://docs.pytest.org/en/latest/example/simple.html#control-skipping-of-tests-according-to-command-line-option
    for details.
    """
    skip_gui = pytest.mark.skip(reason='need X server (Linux only) to run GUI tests')
    for item in items:
        if 'gui' in item.location[0] and hasattr(os, 'uname') and os.uname().sysname != 'Linux':
            item.add_marker(skip_gui)


# === Backend stuff

def get_test_wrapper() -> MatrixWrapper:
    """Return a new MatrixWrapper object with some preset values."""
    wrapper = MatrixWrapper()

    root_two_over_two = np.sqrt(2) / 2

    wrapper['A'] = np.array([[1, 2], [3, 4]])
    wrapper['B'] = np.array([[6, 4], [12, 9]])
    wrapper['C'] = np.array([[-1, -3], [4, -12]])
    wrapper['D'] = np.array([[13.2, 9.4], [-3.4, -1.8]])
    wrapper['E'] = np.array([
        [root_two_over_two, -1 * root_two_over_two],
        [root_two_over_two, root_two_over_two]
    ])
    wrapper['F'] = np.array([[-1, 0], [0, 1]])
    wrapper['G'] = np.array([[np.pi, np.e], [1729, 743.631]])

    return wrapper


@pytest.fixture
def test_wrapper() -> MatrixWrapper:
    """Return a new MatrixWrapper object with some preset values."""
    return get_test_wrapper()


@pytest.fixture
```

```
68      def new_wrapper() -> MatrixWrapper:
69          """Return a new MatrixWrapper with no initialized values."""
70          return MatrixWrapper()
71
72
73      # === GUI stuff
74
75      def is_widget_class_open(widget_class: Type[QWidget]) -> bool:
76          """Test if a widget with the given class is currently open."""
77          return widget_class in [x.__class__ for x in QApplication.topLevelWidgets()]
78
79
80      @pytest.fixture
81      def window(qtbot: QtBot) -> LintransMainWindow:
82          """Return an instance of :class:`LintransMainWindow`."""
83          window = LintransMainWindow()
84          qtbot.addWidget(window)
85          return window
86
87
88      def get_open_widget(widget_class: Type[T]) -> T:
89          """Get the open instance of the given :class:`QWidget` subclass.
90
91          This method assumes that there is exactly 1 widget of the given
92          class and will raise ``ValueError`` if there's not.
93
94          :raises ValueError: If there is not exactly one widget of the given class
95          """
96          widgets = [
97              x for x in QApplication.topLevelWidgets()
98              if isinstance(x, widget_class)
99          ]
100
101         if len(widgets) != 1:
102             raise ValueError(f'Expected 1 widget of type {widget_class} but found {len(widgets)}')
103
104         return widgets[0]
```

B.2 gui/test_define_dialogs.py

```
1       # lintrans - The linear transformation visualizer
2       # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4       # This program is licensed under GNU GPLv3, available here:
5       # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7       """Test the :class:`DefineDialog` boxes in :class:`LintransMainWindow`."""
8
9       import numpy as np
10      from PyQt5.QtCore import Qt
11      from pytestqt.qtbot import QtBot
12
13      from lintrans.gui.dialogs import DefineAsExpressionDialog, DefineNumericallyDialog, DefineVisuallyDialog
14      from lintrans.gui.main_window import LintransMainWindow
15
16      from conftest import get_open_widget, is_widget_class_open
17
18      ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
19
20
21      def test_define_visually_dialog_opens(qtbot: QtBot, window: LintransMainWindow) -> None:
22          """Test that the :class:`DefineVisuallyDialog` opens."""
23          qtbot.mouseClick(window._button_define_visually, Qt.LeftButton)
24          assert is_widget_class_open(DefineVisuallyDialog)
25          qtbot.addWidget(get_open_widget(DefineVisuallyDialog))
26
27
28      def test_define_numerically_dialog_opens(qtbot: QtBot, window: LintransMainWindow) -> None:
29          """Test that the :class:`DefineNumericallyDialog` opens."""
30          qtbot.mouseClick(window._button_define_numerically, Qt.LeftButton)
```

```
        assert is_widget_class_open(DefineNumericallyDialog)
        qtbot.addWidget(get_open_widget(DefineNumericallyDialog))


    def test_define_as_expression_dialog_opens(qtbot: QtBot, window: LintransMainWindow) -> None:
        """Test that the :class:`DefineAsAnExpressionDialog` opens."""
        qtbot.mouseClick(window._button_define_as_expression, Qt.LeftButton)
        assert is_widget_class_open(DefineAsExpressionDialog)
        qtbot.addWidget(get_open_widget(DefineAsExpressionDialog))


    def test_define_numerically_dialog_works(qtbot: QtBot, window: LintransMainWindow) -> None:
        """Test that matrices can be defined numerically."""
        qtbot.mouseClick(window._button_define_numerically, Qt.LeftButton)
        dialog = get_open_widget(DefineNumericallyDialog)
        qtbot.addWidget(dialog)

        qtbot.keyClicks(dialog._element_tl, '-1')
        qtbot.keyClicks(dialog._element_tr, '3')
        qtbot.keyClicks(dialog._element_bl, '2')
        qtbot.keyClicks(dialog._element_br, '-0.5')

        qtbot.mouseClick(dialog._button_confirm, Qt.LeftButton)

        assert (window._matrix_wrapper['A'] == np.array([
            [-1, 3],
            [2, -0.5]
        ])).all()


    def test_define_as_expression_dialog_works(qtbot: QtBot, window: LintransMainWindow) -> None:
        """Test that matrices can be defined as expressions."""
        qtbot.mouseClick(window._button_define_as_expression, Qt.LeftButton)
        dialog = get_open_widget(DefineAsExpressionDialog)
        qtbot.addWidget(dialog)

        qtbot.keyClicks(dialog._lineedit_expression_box, '(rot(45)^{2}3I)^Trot(210)^-1')
        qtbot.mouseClick(dialog._button_confirm, Qt.LeftButton)

        assert window._matrix_wrapper.get_expression('A') == '(rot(45)^{2}3I)^Trot(210)^-1'
        assert (
            window._matrix_wrapper['A'] ==
            window._matrix_wrapper.evaluate_expression('(rot(45)^{2}3I)^Trot(210)^-1')
        ).all()
```

B.3 gui/test_other_dialogs.py

```
    # lintrans - The linear transformation visualizer
    # Copyright (C) 2022 D. Dyson (DoctorDalek1963)
    #
    # This program is licensed under GNU GPLv3, available here:
    # <https://www.gnu.org/licenses/gpl-3.0.html>

    """Test that the non-defintion dialogs work as expected."""

    from typing import Type

    import pytest
    from PyQt5.QtCore import Qt
    from PyQt5.QtWidgets import QDialog
    from pytestqt.qtbot import QtBot

    from lintrans.gui.dialogs import DisplaySettingsDialog, InfoPanelDialog
    from lintrans.gui.main_window import LintransMainWindow

    from conftest import get_open_widget, is_widget_class_open


    @pytest.mark.parametrize(
        'button_attr,dialog_class',
```

```
        [
            ('_button_change_display_settings', DisplaySettingsDialog),
            ('_button_info_panel', InfoPanelDialog),
        ]
    )
    def test_dialogs_open(
        qtbot: QtBot,
        window: LintransMainWindow,
        button_attr: str,
        dialog_class: Type[QDialog]
    ) -> None:
        """Make sure the dialog opens properly."""
        qtbot.mouseClick(getattr(window, button_attr), Qt.LeftButton)
        assert is_widget_class_open(dialog_class)
        qtbot.addWidget(get_open_widget(dialog_class))
```

B.4 backend/test_session.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""Test the functionality of saving and loading sessions."""

from pathlib import Path

import lintrans
from lintrans.gui.session import Session
from lintrans.matrices.wrapper import MatrixWrapper

from conftest import get_test_wrapper


def test_save_and_load(tmp_path: Path, test_wrapper: MatrixWrapper) -> None:
    """Test that sessions save and load and return the same matrix wrapper."""
    points = [(1, 0), (-2, 3), (3.2, -10), (0, 0), (-2, -3), (2, -1.3)]
    session = Session(matrix_wrapper=test_wrapper, polygon_points=points)

    path = str((tmp_path / 'test.lt').absolute())
    session.save_to_file(path)

    loaded_session, version, extra_attrs = Session.load_from_file(path)
    assert loaded_session.matrix_wrapper == get_test_wrapper()
    assert loaded_session.polygon_points == points

    assert version == lintrans.__version__
    assert not extra_attrs
```

B.5 backend/matrices/test_parse_and_validate_expression.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""Test the :mod:`matrices.parse` module validation and parsing."""

from typing import List, Tuple

import pytest

from lintrans.matrices.parse import (MatrixParseError, find_sub_expressions, get_matrix_identifiers,
                                     parse_matrix_expression, validate_matrix_expression)
from lintrans.typing_ import MatrixParseList

```

```
    expected_sub_expressions: List[Tuple[str, List[str]]] = [
        ('2(AB)^-1', ['AB']),
        ('-3(A+B)^2-C(B^TA)^-1', ['A+B', 'B^TA']),
        ('rot(45)', []),
        ('()', []),
        ('(())', ['()']),
        ('2.3A^-1(AB)^-1+(BC)^2', ['AB', 'BC']),
        ('(2.3A^-1(AB)^-1+(BC)^2)', ['2.3A^-1(AB)^-1+(BC)^2']),
    ]


    def test_find_sub_expressions() -> None:
        """Test the :func:`lintrans.matrices.parse.find_sub_expressions` function."""
        for inp, output in expected_sub_expressions:
            assert find_sub_expressions(inp) == output


    valid_inputs: List[str] = [
        'A', 'AB', '3A', '1.2A', '-3.4A', 'A^2', 'A^-1', 'A^{-1}',
        'A^12', 'A^T', 'A^{5}', 'A^{T}', '4.3A^7', '9.2A^{18}', '0.1A'

        'rot(45)', 'rot(12.5)', '3rot(90)',
        'rot(135)^3', 'rot(51)^T', 'rot(-34)^-1',

        'A+B', 'A+2B', '4.3A+9B', 'A^2+B^T', '3A^7+0.8B^{16}',
        'A-B', '3A-4B', '3.2A^3-16.79B^T', '4.752A^{17}-3.32B^{36}',
        'A-1B', '-A', '-1A', 'A^{2}3.4B', 'A^{-1}2.3B',

        '3A4B', 'A^TB', 'A^{T}B', '4A^6B^3',
        '2A^{3}4B^5', '4rot(90)^3', 'rot(45)rot(13)',
        'Arot(90)', 'AB^2', 'A^2B^2', '8.36A^T3.4B^12',

        '3.5A^{4}5.6rot(19.2)^T-B^{-1}4.1C^5',

        '(A)', '(AB)^-1', '2.3(3B^TA)^2', '-3.4(9D^{2}3F^-1)^T+C', '(AB)(C)',
        '3(rot(34)^-7A)^-1+B', '3A^2B+4A(B+C)^-1D^T-A(C(D+E)B)'
    ]

    invalid_inputs: List[str] = [
        '', 'rot()', 'A^', 'A^1.2', 'A^2 3.4B', 'A^23.4B', 'A^-1 2.3B', 'A^{3.4}', '1,2A', 'ro(12)', '5', '12^2',
        '^T', '^{12}', '.1A', 'A^{13', 'A^3}', 'A^A', '^2', 'A--B', '--A', '+A', '--1A', 'A--B', 'A--1B',
        '.A', '1.A', '2.3AB)^T', '(AB+)', '-4.6(9A', '-2(3.4A^{-1}-C^)^2', '9.2)', '3A^2B+4A(B+C)^-1D^T-A(C(D+EB)',
        '3()^2', '4(your mum)^T', 'rot()', 'rot(10.1.1)', 'rot(--2)',

        'This is 100% a valid matrix expression, I swear'
    ]


    @pytest.mark.parametrize('inputs,output', [(valid_inputs, True), (invalid_inputs, False)])
    def test_validate_matrix_expression(inputs: List[str], output: bool) -> None:
        """Test the validate_matrix_expression() function."""
        for inp in inputs:
            assert validate_matrix_expression(inp) == output


    expressions_and_parsed_expressions: List[Tuple[str, MatrixParseList]] = [
        # Simple expressions
        ('A', [[('', 'A', '')]]),
        ('A^2', [[('', 'A', '2')]]),
        ('A^{2}', [[('', 'A', '2')]]),
        ('3A', [[('3', 'A', '')]]),
        ('1.4A^3', [[('1.4', 'A', '3')]]),
        ('0.1A', [[('0.1', 'A', '')]]),
        ('0.1A', [[('0.1', 'A', '')]]),
        ('A^12', [[('', 'A', '12')]]),
        ('A^234', [[('', 'A', '234')]]),

        # Multiplications
        ('A 0.1B', [[('', 'A', ''), ('0.1', 'B', '')]]),
        ('A^2 3B', [[('', 'A', '23'), ('', 'B', '')]]),
        ('A^{2}3.4B', [[('', 'A', '2'), ('3.4', 'B', '')]]),
        ('4A^{3} 6B^2', [[('4', 'A', '3'), ('6', 'B', '2')]]),
        ('4.2A^{T} 6.1B^-1', [[('4.2', 'A', 'T'), ('6.1', 'B', '-1')]]),
```

```
        ('-1.2A^2 rot(45)^2', [[('-1.2', 'A', '2'), ('', 'rot(45)', '2')]]),
        ('3.2A^T 4.5B^{5} 9.6rot(121.3)', [[('3.2', 'A', 'T'), ('4.5', 'B', '5'), ('9.6', 'rot(121.3)', '')]]),
        ('-1.18A^{-2} 0.1B^{2} 9rot(-34.6)^-1', [[('-1.18', 'A', '-2'), ('0.1', 'B', '2'), ('9', 'rot(-34.6)', '-1')]]),

        # Additions
        ('A + B', [[('', 'A', '')], [('', 'B', '')]]),
        ('A + B - C', [[('', 'A', '')], [('', 'B', '')], [('-1', 'C', '')]]),
        ('A^2 + 0.5B', [[('', 'A', '2')], [('0.5', 'B', '')]]),
        ('2A^3 + 8B^T - 3C^-1', [[('2', 'A', '3')], [('8', 'B', 'T')], [('-3', 'C', '-1')]]),
        ('4.9A^2 - 3rot(134.2)^-1 + 7.6B^8', [[('4.9', 'A', '2')], [('-3', 'rot(134.2)', '-1')], [('7.6', 'B', '8')]]),

        # Additions with multiplication
        ('2.14A^{3} 4.5rot(14.5)^-1 + 8B^T - 3C^-1', [[('2.14', 'A', '3'), ('4.5', 'rot(14.5)', '-1')],
                                                      [('8', 'B', 'T')], [('-3', 'C', '-1')]]),
        ('2.14A^{3} 4.5rot(14.5)^-1 + 8.5B^T 5.97C^14 - 3.14D^{-1} 6.7E^T',
         [[('2.14', 'A', '3'), ('4.5', 'rot(14.5)', '-1')], [('8.5', 'B', 'T'), ('5.97', 'C', '14')],
          [('-3.14', 'D', '-1'), ('6.7', 'E', 'T')]]),

        # Parenthesized expressions
        ('(AB)^-1', [[('', 'AB', '-1')]]),
        ('-3(A+B)^2-C(B^TA)^-1', [[('-3', 'A+B', '2')], [('-1', 'C', ''), ('', 'B^{T}A', '-1')]]),
        ('2.3(3B^TA)^2', [[('2.3', '3B^{T}A', '2')]]),
        ('-3.4(9D^{2}3F^-1)^T+C', [[('-3.4', '9D^{2}3F^{-1}', 'T')], [('', 'C', '')]]),
        ('2.39(3.1A^{-1}2.3B(CD)^-1)^T + (AB^T)^-1', [[('2.39', '3.1A^{-1}2.3B(CD)^{-1}', 'T')], [('', 'AB^{T}',
        ↪  '-1')]])
    ]


    def test_parse_matrix_expression() -> None:
        """Test the parse_matrix_expression() function."""
        for expression, parsed_expression in expressions_and_parsed_expressions:
            # Test it with and without whitespace
            assert parse_matrix_expression(expression) == parsed_expression
            assert parse_matrix_expression(expression.replace(' ', '')) == parsed_expression

        for expression in valid_inputs:
            # Assert that it doesn't raise MatrixParseError
            parse_matrix_expression(expression)


    def test_parse_error() -> None:
        """Test that parse_matrix_expression() raises a MatrixParseError."""
        for expression in invalid_inputs:
            with pytest.raises(MatrixParseError):
                parse_matrix_expression(expression)


    def test_get_matrix_identifiers() -> None:
        """Test that matrix identifiers can be properly found."""
        assert get_matrix_identifiers('M^T') == {'M'}
        assert get_matrix_identifiers('ABCDEF') == {'A', 'B', 'C', 'D', 'E', 'F'}
        assert get_matrix_identifiers('AB^{-1}3Crot(45)2A(B^2C^-1)') == {'A', 'B', 'C'}
        assert get_matrix_identifiers('A^{2}3A^-1A^TA') == {'A'}
        assert get_matrix_identifiers('rot(45)(rot(25)rot(20))^2') == set()

        for expression in invalid_inputs:
            with pytest.raises(MatrixParseError):
                get_matrix_identifiers(expression)
```

B.6 backend/matrices/matrix_wrapper/test_evaluate_expression.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""Test the MatrixWrapper evaluate_expression() method."""

import numpy as np
```

```python
from numpy import linalg as la
import pytest
from pytest import approx

from lintrans.matrices import MatrixWrapper, create_rotation_matrix
from lintrans.typing_ import MatrixType

from conftest import get_test_wrapper


def test_simple_matrix_addition(test_wrapper: MatrixWrapper) -> None:
    """Test simple addition and subtraction of two matrices."""

    # NOTE: We assert that all of these values are not None just to stop mypy complaining
    # These values will never actually be None because they're set in the wrapper() fixture
    # There's probably a better way  do this, because this method is a bit of a bodge, but this works for now
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

    assert (test_wrapper.evaluate_expression('A+B') == test_wrapper['A'] + test_wrapper['B']).all()
    assert (test_wrapper.evaluate_expression('E+F') == test_wrapper['E'] + test_wrapper['F']).all()
    assert (test_wrapper.evaluate_expression('G+D') == test_wrapper['G'] + test_wrapper['D']).all()
    assert (test_wrapper.evaluate_expression('C+C') == test_wrapper['C'] + test_wrapper['C']).all()
    assert (test_wrapper.evaluate_expression('D+A') == test_wrapper['D'] + test_wrapper['A']).all()
    assert (test_wrapper.evaluate_expression('B+C') == test_wrapper['B'] + test_wrapper['C']).all()

    assert test_wrapper == get_test_wrapper()


def test_simple_two_matrix_multiplication(test_wrapper: MatrixWrapper) -> None:
    """Test simple multiplication of two matrices."""
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

    assert (test_wrapper.evaluate_expression('AB') == test_wrapper['A'] @ test_wrapper['B']).all()
    assert (test_wrapper.evaluate_expression('BA') == test_wrapper['B'] @ test_wrapper['A']).all()
    assert (test_wrapper.evaluate_expression('AC') == test_wrapper['A'] @ test_wrapper['C']).all()
    assert (test_wrapper.evaluate_expression('DA') == test_wrapper['D'] @ test_wrapper['A']).all()
    assert (test_wrapper.evaluate_expression('ED') == test_wrapper['E'] @ test_wrapper['D']).all()
    assert (test_wrapper.evaluate_expression('FD') == test_wrapper['F'] @ test_wrapper['D']).all()
    assert (test_wrapper.evaluate_expression('GA') == test_wrapper['G'] @ test_wrapper['A']).all()
    assert (test_wrapper.evaluate_expression('CF') == test_wrapper['C'] @ test_wrapper['F']).all()
    assert (test_wrapper.evaluate_expression('AG') == test_wrapper['A'] @ test_wrapper['G']).all()

    assert test_wrapper.evaluate_expression('A2B') == approx(test_wrapper['A'] @ (2 * test_wrapper['B']))
    assert test_wrapper.evaluate_expression('2AB') == approx((2 * test_wrapper['A']) @ test_wrapper['B'])
    assert test_wrapper.evaluate_expression('C3D') == approx(test_wrapper['C'] @ (3 * test_wrapper['D']))
    assert test_wrapper.evaluate_expression('4.2E1.2A') == approx((4.2 * test_wrapper['E']) @ (1.2 *
    ↪  test_wrapper['A']))

    assert test_wrapper == get_test_wrapper()


def test_identity_multiplication(test_wrapper: MatrixWrapper) -> None:
    """Test that multiplying by the identity doesn't change the value of a matrix."""
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

    assert (test_wrapper.evaluate_expression('I') == test_wrapper['I']).all()
    assert (test_wrapper.evaluate_expression('AI') == test_wrapper['A']).all()
    assert (test_wrapper.evaluate_expression('IA') == test_wrapper['A']).all()
    assert (test_wrapper.evaluate_expression('GI') == test_wrapper['G']).all()
    assert (test_wrapper.evaluate_expression('IG') == test_wrapper['G']).all()

    assert (test_wrapper.evaluate_expression('EID') == test_wrapper['E'] @ test_wrapper['D']).all()
    assert (test_wrapper.evaluate_expression('IED') == test_wrapper['E'] @ test_wrapper['D']).all()
    assert (test_wrapper.evaluate_expression('EDI') == test_wrapper['E'] @ test_wrapper['D']).all()
    assert (test_wrapper.evaluate_expression('IEIDI') == test_wrapper['E'] @ test_wrapper['D']).all()
    assert (test_wrapper.evaluate_expression('EI^3D') == test_wrapper['E'] @ test_wrapper['D']).all()

```

```python
    assert test_wrapper == get_test_wrapper()


def test_simple_three_matrix_multiplication(test_wrapper: MatrixWrapper) -> None:
    """Test simple multiplication of two matrices."""
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

    assert (test_wrapper.evaluate_expression('ABC') == test_wrapper['A'] @ test_wrapper['B'] @ test_wrapper['C']).all()
    assert (test_wrapper.evaluate_expression('ACB') == test_wrapper['A'] @ test_wrapper['C'] @ test_wrapper['B']).all()
    assert (test_wrapper.evaluate_expression('BAC') == test_wrapper['B'] @ test_wrapper['A'] @ test_wrapper['C']).all()
    assert (test_wrapper.evaluate_expression('EFG') == test_wrapper['E'] @ test_wrapper['F'] @ test_wrapper['G']).all()
    assert (test_wrapper.evaluate_expression('DAC') == test_wrapper['D'] @ test_wrapper['A'] @ test_wrapper['C']).all()
    assert (test_wrapper.evaluate_expression('GAE') == test_wrapper['G'] @ test_wrapper['A'] @ test_wrapper['E']).all()
    assert (test_wrapper.evaluate_expression('FAG') == test_wrapper['F'] @ test_wrapper['A'] @ test_wrapper['G']).all()
    assert (test_wrapper.evaluate_expression('GAF') == test_wrapper['G'] @ test_wrapper['A'] @ test_wrapper['F']).all()

    assert test_wrapper == get_test_wrapper()


def test_matrix_inverses(test_wrapper: MatrixWrapper) -> None:
    """Test the inverses of single matrices."""
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

    assert (test_wrapper.evaluate_expression('A^{-1}') == la.inv(test_wrapper['A'])).all()
    assert (test_wrapper.evaluate_expression('B^{-1}') == la.inv(test_wrapper['B'])).all()
    assert (test_wrapper.evaluate_expression('C^{-1}') == la.inv(test_wrapper['C'])).all()
    assert (test_wrapper.evaluate_expression('D^{-1}') == la.inv(test_wrapper['D'])).all()
    assert (test_wrapper.evaluate_expression('E^{-1}') == la.inv(test_wrapper['E'])).all()
    assert (test_wrapper.evaluate_expression('F^{-1}') == la.inv(test_wrapper['F'])).all()
    assert (test_wrapper.evaluate_expression('G^{-1}') == la.inv(test_wrapper['G'])).all()

    assert (test_wrapper.evaluate_expression('A^-1') == la.inv(test_wrapper['A'])).all()
    assert (test_wrapper.evaluate_expression('B^-1') == la.inv(test_wrapper['B'])).all()
    assert (test_wrapper.evaluate_expression('C^-1') == la.inv(test_wrapper['C'])).all()
    assert (test_wrapper.evaluate_expression('D^-1') == la.inv(test_wrapper['D'])).all()
    assert (test_wrapper.evaluate_expression('E^-1') == la.inv(test_wrapper['E'])).all()
    assert (test_wrapper.evaluate_expression('F^-1') == la.inv(test_wrapper['F'])).all()
    assert (test_wrapper.evaluate_expression('G^-1') == la.inv(test_wrapper['G'])).all()

    assert test_wrapper == get_test_wrapper()


def test_matrix_powers(test_wrapper: MatrixWrapper) -> None:
    """Test that matrices can be raised to integer powers."""
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

    assert (test_wrapper.evaluate_expression('A^2') == la.matrix_power(test_wrapper['A'], 2)).all()
    assert (test_wrapper.evaluate_expression('B^4') == la.matrix_power(test_wrapper['B'], 4)).all()
    assert (test_wrapper.evaluate_expression('C^{12}') == la.matrix_power(test_wrapper['C'], 12)).all()
    assert (test_wrapper.evaluate_expression('D^12') == la.matrix_power(test_wrapper['D'], 12)).all()
    assert (test_wrapper.evaluate_expression('E^8') == la.matrix_power(test_wrapper['E'], 8)).all()
    assert (test_wrapper.evaluate_expression('F^{-6}') == la.matrix_power(test_wrapper['F'], -6)).all()
    assert (test_wrapper.evaluate_expression('G^-2') == la.matrix_power(test_wrapper['G'], -2)).all()

    assert test_wrapper == get_test_wrapper()


def test_matrix_transpose(test_wrapper: MatrixWrapper) -> None:
    """Test matrix transpositions."""
```

```
147         assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
148                test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
149                test_wrapper['G'] is not None
150
151         assert (test_wrapper.evaluate_expression('A^{T}') == test_wrapper['A'].T).all()
152         assert (test_wrapper.evaluate_expression('B^{T}') == test_wrapper['B'].T).all()
153         assert (test_wrapper.evaluate_expression('C^{T}') == test_wrapper['C'].T).all()
154         assert (test_wrapper.evaluate_expression('D^{T}') == test_wrapper['D'].T).all()
155         assert (test_wrapper.evaluate_expression('E^{T}') == test_wrapper['E'].T).all()
156         assert (test_wrapper.evaluate_expression('F^{T}') == test_wrapper['F'].T).all()
157         assert (test_wrapper.evaluate_expression('G^{T}') == test_wrapper['G'].T).all()
158
159         assert (test_wrapper.evaluate_expression('A^T') == test_wrapper['A'].T).all()
160         assert (test_wrapper.evaluate_expression('B^T') == test_wrapper['B'].T).all()
161         assert (test_wrapper.evaluate_expression('C^T') == test_wrapper['C'].T).all()
162         assert (test_wrapper.evaluate_expression('D^T') == test_wrapper['D'].T).all()
163         assert (test_wrapper.evaluate_expression('E^T') == test_wrapper['E'].T).all()
164         assert (test_wrapper.evaluate_expression('F^T') == test_wrapper['F'].T).all()
165         assert (test_wrapper.evaluate_expression('G^T') == test_wrapper['G'].T).all()
166
167         assert test_wrapper == get_test_wrapper()
168
169
170     def test_rotation_matrices(test_wrapper: MatrixWrapper) -> None:
171         """Test that 'rot(angle)' can be used in an expression."""
172         assert (test_wrapper.evaluate_expression('rot(90)') == create_rotation_matrix(90)).all()
173         assert (test_wrapper.evaluate_expression('rot(180)') == create_rotation_matrix(180)).all()
174         assert (test_wrapper.evaluate_expression('rot(270)') == create_rotation_matrix(270)).all()
175         assert (test_wrapper.evaluate_expression('rot(360)') == create_rotation_matrix(360)).all()
176         assert (test_wrapper.evaluate_expression('rot(45)') == create_rotation_matrix(45)).all()
177         assert (test_wrapper.evaluate_expression('rot(30)') == create_rotation_matrix(30)).all()
178
179         assert (test_wrapper.evaluate_expression('rot(13.43)') == create_rotation_matrix(13.43)).all()
180         assert (test_wrapper.evaluate_expression('rot(49.4)') == create_rotation_matrix(49.4)).all()
181         assert (test_wrapper.evaluate_expression('rot(-123.456)') == create_rotation_matrix(-123.456)).all()
182         assert (test_wrapper.evaluate_expression('rot(963.245)') == create_rotation_matrix(963.245)).all()
183         assert (test_wrapper.evaluate_expression('rot(-235.24)') == create_rotation_matrix(-235.24)).all()
184
185         assert test_wrapper == get_test_wrapper()
186
187
188     def test_multiplication_and_addition(test_wrapper: MatrixWrapper) -> None:
189         """Test multiplication and addition of matrices together."""
190         assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
191                test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
192                test_wrapper['G'] is not None
193
194         assert (test_wrapper.evaluate_expression('AB+C') ==
195                 test_wrapper['A'] @ test_wrapper['B'] + test_wrapper['C']).all()
196         assert (test_wrapper.evaluate_expression('DE-D') ==
197                 test_wrapper['D'] @ test_wrapper['E'] - test_wrapper['D']).all()
198         assert (test_wrapper.evaluate_expression('FD+AB') ==
199                 test_wrapper['F'] @ test_wrapper['D'] + test_wrapper['A'] @ test_wrapper['B']).all()
200         assert (test_wrapper.evaluate_expression('BA-DE') ==
201                 test_wrapper['B'] @ test_wrapper['A'] - test_wrapper['D'] @ test_wrapper['E']).all()
202
203         assert (test_wrapper.evaluate_expression('2AB+3C') ==
204                 (2 * test_wrapper['A']) @ test_wrapper['B'] + (3 * test_wrapper['C'])).all()
205         assert (test_wrapper.evaluate_expression('4D7.9E-1.2A') ==
206                 (4 * test_wrapper['D']) @ (7.9 * test_wrapper['E']) - (1.2 * test_wrapper['A'])).all()
207
208         assert test_wrapper == get_test_wrapper()
209
210
211     def test_complicated_expressions(test_wrapper: MatrixWrapper) -> None:
212         """Test evaluation of complicated expressions."""
213         assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
214                test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
215                test_wrapper['G'] is not None
216
217         assert (test_wrapper.evaluate_expression('-3.2A^T 4B^{-1} 6C^{-1} + 8.1D^{2} 3.2E^4') ==
218                 (-3.2 * test_wrapper['A'].T) @ (4 * la.inv(test_wrapper['B'])) @ (6 * la.inv(test_wrapper['C']))
219                 + (8.1 * la.matrix_power(test_wrapper['D'], 2)) @ (3.2 * la.matrix_power(test_wrapper['E'], 4))).all()
```

```python

    assert (test_wrapper.evaluate_expression('53.6D^{2} 3B^T - 4.9F^{2} 2D + A^3 B^-1') ==
            (53.6 * la.matrix_power(test_wrapper['D'], 2)) @ (3 * test_wrapper['B'].T)
            - (4.9 * la.matrix_power(test_wrapper['F'], 2)) @ (2 * test_wrapper['D'])
            + la.matrix_power(test_wrapper['A'], 3) @ la.inv(test_wrapper['B'])).all()

    assert test_wrapper == get_test_wrapper()


def test_parenthesized_expressions(test_wrapper: MatrixWrapper) -> None:
    """Test evaluation of parenthesized expressions."""
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

    assert (test_wrapper.evaluate_expression('(A^T)^2') == la.matrix_power(test_wrapper['A'].T, 2)).all()
    assert (test_wrapper.evaluate_expression('(B^T)^3') == la.matrix_power(test_wrapper['B'].T, 3)).all()
    assert (test_wrapper.evaluate_expression('(C^T)^4') == la.matrix_power(test_wrapper['C'].T, 4)).all()
    assert (test_wrapper.evaluate_expression('(D^T)^5') == la.matrix_power(test_wrapper['D'].T, 5)).all()
    assert (test_wrapper.evaluate_expression('(E^T)^6') == la.matrix_power(test_wrapper['E'].T, 6)).all()
    assert (test_wrapper.evaluate_expression('(F^T)^7') == la.matrix_power(test_wrapper['F'].T, 7)).all()
    assert (test_wrapper.evaluate_expression('(G^T)^8') == la.matrix_power(test_wrapper['G'].T, 8)).all()

    assert (test_wrapper.evaluate_expression('(rot(45)^1)^T') == create_rotation_matrix(45).T).all()
    assert (test_wrapper.evaluate_expression('(rot(45)^2)^T') == la.matrix_power(create_rotation_matrix(45),
    ↪  2).T).all()
    assert (test_wrapper.evaluate_expression('(rot(45)^3)^T') == la.matrix_power(create_rotation_matrix(45),
    ↪  3).T).all()
    assert (test_wrapper.evaluate_expression('(rot(45)^4)^T') == la.matrix_power(create_rotation_matrix(45),
    ↪  4).T).all()
    assert (test_wrapper.evaluate_expression('(rot(45)^5)^T') == la.matrix_power(create_rotation_matrix(45),
    ↪  5).T).all()

    assert (test_wrapper.evaluate_expression('D^3(A+6.2F-0.397G^TE)^-2+A') ==
            la.matrix_power(test_wrapper['D'], 3) @ la.matrix_power(
                test_wrapper['A'] + 6.2 * test_wrapper['F'] - 0.397 * test_wrapper['G'].T @ test_wrapper['E'],
                -2
            ) + test_wrapper['A']).all()

    assert (test_wrapper.evaluate_expression('-1.2F^{3}4.9D^T(A^2(B+3E^TF)^-1)^2') ==
            -1.2 * la.matrix_power(test_wrapper['F'], 3) @ (4.9 * test_wrapper['D'].T) @
            la.matrix_power(
                la.matrix_power(test_wrapper['A'], 2) @ la.matrix_power(
                    test_wrapper['B'] + 3 * test_wrapper['E'].T @ test_wrapper['F'],
                    -1
                ),
                2
            )).all()


def test_value_errors(test_wrapper: MatrixWrapper) -> None:
    """Test that evaluate_expression() raises a ValueError for any malformed input."""
    invalid_expressions = ['', '+', '-', 'This is not a valid expression', '3+4',
                           'A+2', 'A^', '^2', 'A^-', 'At', 'A^t', '3^2']

    for expression in invalid_expressions:
        with pytest.raises(ValueError):
            test_wrapper.evaluate_expression(expression)


def test_linalgerror() -> None:
    """Test that certain expressions raise np.linalg.LinAlgError."""
    matrix_a: MatrixType = np.array([
        [0, 0],
        [0, 0]
    ])

    matrix_b: MatrixType = np.array([
        [1, 2],
        [1, 2]
    ])

    wrapper = MatrixWrapper()
```

```
        wrapper['A'] = matrix_a
        wrapper['B'] = matrix_b

        assert (wrapper.evaluate_expression('A') == matrix_a).all()
        assert (wrapper.evaluate_expression('B') == matrix_b).all()

        with pytest.raises(np.linalg.LinAlgError):
            wrapper.evaluate_expression('A^-1')

        with pytest.raises(np.linalg.LinAlgError):
            wrapper.evaluate_expression('B^-1')

        assert (wrapper['A'] == matrix_a).all()
        assert (wrapper['B'] == matrix_b).all()
```

B.7 backend/matrices/matrix_wrapper/test_setting_and_getting.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""Test the MatrixWrapper __setitem__() and __getitem__() methods."""

from typing import Any, Dict, List

import numpy as np
import pytest
from numpy import linalg as la

from lintrans.matrices import MatrixWrapper
from lintrans.typing_ import MatrixType

valid_matrix_names = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
invalid_matrix_names = ['bad name', '123456', 'Th15 Is an 1nV@l1D n@m3', 'abc', 'a']

test_matrix: MatrixType = np.array([[1, 2], [4, 3]])


def test_basic_get_matrix(new_wrapper: MatrixWrapper) -> None:
    """Test MatrixWrapper().__getitem__()."""
    for name in valid_matrix_names:
        assert new_wrapper[name] is None

    assert (new_wrapper['I'] == np.array([[1, 0], [0, 1]])).all()


def test_get_name_error(new_wrapper: MatrixWrapper) -> None:
    """Test that MatrixWrapper().__getitem__() raises a NameError if called with an invalid name."""
    for name in invalid_matrix_names:
        with pytest.raises(NameError):
            _ = new_wrapper[name]


def test_basic_set_matrix(new_wrapper: MatrixWrapper) -> None:
    """Test MatrixWrapper().__setitem__()."""
    for name in valid_matrix_names:
        new_wrapper[name] = test_matrix
        assert (new_wrapper[name] == test_matrix).all()

        new_wrapper[name] = None
        assert new_wrapper[name] is None


def test_set_expression(test_wrapper: MatrixWrapper) -> None:
    """Test that MatrixWrapper.__setitem__() can accept a valid expression."""
    test_wrapper['N'] = 'A^2'
    test_wrapper['O'] = 'BA+2C'
    test_wrapper['P'] = 'E^T'
```

```
        test_wrapper['Q'] = 'C^-1B'
        test_wrapper['R'] = 'A^{2}3B'
        test_wrapper['S'] = 'N^-1'
        test_wrapper['T'] = 'PQP^-1'

        with pytest.raises(TypeError):
            test_wrapper['U'] = 'A+1'

        with pytest.raises(TypeError):
            test_wrapper['V'] = 'K'

        with pytest.raises(TypeError):
            test_wrapper['W'] = 'L^2'

        with pytest.raises(TypeError):
            test_wrapper['X'] = 'M^-1'

        with pytest.raises(TypeError):
            test_wrapper['Y'] = 'A^2B+C^'


    def test_simple_dynamic_evaluation(test_wrapper: MatrixWrapper) -> None:
        """Test that expression-defined matrices are evaluated dynamically."""
        test_wrapper['N'] = 'A^2'
        test_wrapper['O'] = '4B'
        test_wrapper['P'] = 'A+C'

        assert (test_wrapper['N'] == test_wrapper.evaluate_expression('A^2')).all()
        assert (test_wrapper['O'] == test_wrapper.evaluate_expression('4B')).all()
        assert (test_wrapper['P'] == test_wrapper.evaluate_expression('A+C')).all()

        assert (test_wrapper.evaluate_expression('N^2 + 3O') ==
                la.matrix_power(test_wrapper.evaluate_expression('A^2'), 2) +
                3 * test_wrapper.evaluate_expression('4B')
                ).all()
        assert (test_wrapper.evaluate_expression('P^-1 - 3NO^2') ==
                la.inv(test_wrapper.evaluate_expression('A+C')) -
                (3 * test_wrapper.evaluate_expression('A^2')) @
                la.matrix_power(test_wrapper.evaluate_expression('4B'), 2)
                ).all()

        test_wrapper['A'] = np.array([
            [19, -21.5],
            [84, 96.572]
        ])
        test_wrapper['B'] = np.array([
            [-0.993, 2.52],
            [1e10, 0]
        ])
        test_wrapper['C'] = np.array([
            [0, 19512],
            [1.414, 19]
        ])

        assert (test_wrapper['N'] == test_wrapper.evaluate_expression('A^2')).all()
        assert (test_wrapper['O'] == test_wrapper.evaluate_expression('4B')).all()
        assert (test_wrapper['P'] == test_wrapper.evaluate_expression('A+C')).all()

        assert (test_wrapper.evaluate_expression('N^2 + 3O') ==
                la.matrix_power(test_wrapper.evaluate_expression('A^2'), 2) +
                3 * test_wrapper.evaluate_expression('4B')
                ).all()
        assert (test_wrapper.evaluate_expression('P^-1 - 3NO^2') ==
                la.inv(test_wrapper.evaluate_expression('A+C')) -
                (3 * test_wrapper.evaluate_expression('A^2')) @
                la.matrix_power(test_wrapper.evaluate_expression('4B'), 2)
                ).all()


    def test_recursive_dynamic_evaluation(test_wrapper: MatrixWrapper) -> None:
        """Test that dynamic evaluation works recursively."""
        test_wrapper['N'] = 'A^2'
        test_wrapper['O'] = '4B'
```

```python
    test_wrapper['P'] = 'A+C'

    test_wrapper['Q'] = 'N^-1'
    test_wrapper['R'] = 'P-4O'
    test_wrapper['S'] = 'NOP'

    assert test_wrapper['Q'] == pytest.approx(test_wrapper.evaluate_expression('A^-2'))
    assert test_wrapper['R'] == pytest.approx(test_wrapper.evaluate_expression('A + C - 16B'))
    assert test_wrapper['S'] == pytest.approx(test_wrapper.evaluate_expression('A^{2}4BA + A^{2}4BC'))


def test_self_referential_expressions(test_wrapper: MatrixWrapper) -> None:
    """Test that self-referential expressions raise an error."""
    expressions: Dict[str, str] = {
        'A': 'A^2',
        'B': 'A(C^-1A^T)+rot(45)B',
        'C': '2Brot(1482.536)(A^-1D^{2}4CE)^3F'
    }

    for name, expression in expressions.items():
        with pytest.raises(ValueError):
            test_wrapper[name] = expression

    test_wrapper['B'] = '3A^2'
    test_wrapper['C'] = 'ABBA'
    with pytest.raises(ValueError):
        test_wrapper['A'] = 'C^-1'

    test_wrapper['E'] = 'rot(45)B^-1+C^T'
    test_wrapper['F'] = 'EBDBIC'
    test_wrapper['D'] = 'E'
    with pytest.raises(ValueError):
        test_wrapper['D'] = 'F'


def test_get_matrix_dependencies(test_wrapper: MatrixWrapper) -> None:
    """Test MatrixWrapper's get_matrix_dependencies() and get_expression_dependencies() methods."""
    test_wrapper['N'] = 'A^2'
    test_wrapper['O'] = '4B'
    test_wrapper['P'] = 'A+C'
    test_wrapper['Q'] = 'N^-1'
    test_wrapper['R'] = 'P-4O'
    test_wrapper['S'] = 'NOP'

    assert test_wrapper.get_matrix_dependencies('A') == set()
    assert test_wrapper.get_matrix_dependencies('B') == set()
    assert test_wrapper.get_matrix_dependencies('C') == set()
    assert test_wrapper.get_matrix_dependencies('D') == set()
    assert test_wrapper.get_matrix_dependencies('E') == set()
    assert test_wrapper.get_matrix_dependencies('F') == set()
    assert test_wrapper.get_matrix_dependencies('G') == set()

    assert test_wrapper.get_matrix_dependencies('N') == {'A'}
    assert test_wrapper.get_matrix_dependencies('O') == {'B'}
    assert test_wrapper.get_matrix_dependencies('P') == {'A', 'C'}
    assert test_wrapper.get_matrix_dependencies('Q') == {'A', 'N'}
    assert test_wrapper.get_matrix_dependencies('R') == {'A', 'B', 'C', 'O', 'P'}
    assert test_wrapper.get_matrix_dependencies('S') == {'A', 'B', 'C', 'N', 'O', 'P'}

    assert test_wrapper.get_expression_dependencies('ABC') == set()
    assert test_wrapper.get_expression_dependencies('NOB') == {'A', 'B'}
    assert test_wrapper.get_expression_dependencies('N^2O^Trot(90)B^-1') == {'A', 'B'}
    assert test_wrapper.get_expression_dependencies('NOP') == {'A', 'B', 'C'}
    assert test_wrapper.get_expression_dependencies('NOPQ') == {'A', 'B', 'C', 'N'}
    assert test_wrapper.get_expression_dependencies('NOPQR') == {'A', 'B', 'C', 'N', 'O', 'P'}
    assert test_wrapper.get_expression_dependencies('NOPQRS') == {'A', 'B', 'C', 'N', 'O', 'P'}


def test_set_identity_error(new_wrapper: MatrixWrapper) -> None:
    """Test that MatrixWrapper().__setitem__() raises a NameError when trying to assign to the identity matrix."""
    with pytest.raises(NameError):
        new_wrapper['I'] = test_matrix

```

```python

def test_set_name_error(new_wrapper: MatrixWrapper) -> None:
    """Test that MatrixWrapper().__setitem__() raises a NameError when trying to assign to an invalid name."""
    for name in invalid_matrix_names:
        with pytest.raises(NameError):
            new_wrapper[name] = test_matrix


def test_set_type_error(new_wrapper: MatrixWrapper) -> None:
    """Test that MatrixWrapper().__setitem__() raises a TypeError when trying to set a non-matrix."""
    invalid_values: List[Any] = [
                                 12,
                                 [1, 2, 3, 4, 5],
                                 [[1, 2], [3, 4]],
                                 True,
                                 24.3222,
                                 'This is totally a matrix, I swear',
                                 MatrixWrapper,
                                 MatrixWrapper(),
                                 np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
                                 np.eye(100)
                                 ]

    for value in invalid_values:
        with pytest.raises(TypeError):
            new_wrapper['M'] = value


def test_get_expression(test_wrapper: MatrixWrapper) -> None:
    """Test the get_expression method of the MatrixWrapper class."""
    test_wrapper['N'] = 'A^2'
    test_wrapper['O'] = '4B'
    test_wrapper['P'] = 'A+C'

    test_wrapper['Q'] = 'N^-1'
    test_wrapper['R'] = 'P-4O'
    test_wrapper['S'] = 'NOP'

    assert test_wrapper.get_expression('A') is None
    assert test_wrapper.get_expression('B') is None
    assert test_wrapper.get_expression('C') is None
    assert test_wrapper.get_expression('D') is None
    assert test_wrapper.get_expression('E') is None
    assert test_wrapper.get_expression('F') is None
    assert test_wrapper.get_expression('G') is None

    assert test_wrapper.get_expression('N') == 'A^2'
    assert test_wrapper.get_expression('O') == '4B'
    assert test_wrapper.get_expression('P') == 'A+C'

    assert test_wrapper.get_expression('Q') == 'N^-1'
    assert test_wrapper.get_expression('R') == 'P-4O'
    assert test_wrapper.get_expression('S') == 'NOP'
```

B.8 backend/matrices/utility/test_coord_conversion.py

```python
# lintrans - The linear transformation visualizer
# Copyright (C) 2022 D. Dyson (DoctorDalek1963)
#
# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""Test conversion between polar and rectilinear coordinates in :mod:`lintrans.matrices.utility`."""

from typing import List, Tuple

from numpy import pi, sqrt
from pytest import approx

from lintrans.matrices.utility import polar_coords, rect_coords
```

```

expected_coords: List[Tuple[Tuple[float, float], Tuple[float, float]]] = [
    ((0, 0), (0, 0)),
    ((1, 1), (sqrt(2), pi / 4)),
    ((0, 1), (1, pi / 2)),
    ((1, 0), (1, 0)),
    ((sqrt(2), sqrt(2)), (2, pi / 4)),
    ((-3, 4), (5, 2.214297436)),
    ((4, -3), (5, 5.639684198)),
    ((5, -0.2), (sqrt(626) / 5, 6.24320662)),
    ((-1.3, -10), (10.08414597, 4.583113976)),
    ((23.4, 0), (23.4, 0)),
    ((pi, -pi), (4.442882938, 1.75 * pi))
]


def test_polar_coords() -> None:
    """Test that :func:`lintrans.matrices.utility.polar_coords` works as expected."""
    for rect, polar in expected_coords:
        assert polar_coords(*rect) == approx(polar)


def test_rect_coords() -> None:
    """Test that :func:`lintrans.matrices.utility.rect_coords` works as expected."""
    for rect, polar in expected_coords:
        assert rect_coords(*polar) == approx(rect)

    assert rect_coords(1, 0) == approx((1, 0))
    assert rect_coords(1, pi) == approx((-1, 0))
    assert rect_coords(1, 2 * pi) == approx((1, 0))
    assert rect_coords(1, 3 * pi) == approx((-1, 0))
    assert rect_coords(1, 4 * pi) == approx((1, 0))
    assert rect_coords(1, 5 * pi) == approx((-1, 0))
    assert rect_coords(1, 6 * pi) == approx((1, 0))
    assert rect_coords(20, 100) == approx(rect_coords(20, 100 % (2 * pi)))
```

B.9 backend/matrices/utility/test_float_utility_functions.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)

# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""Test the utility functions for GUI dialog boxes."""

from typing import List, Tuple

import numpy as np
import pytest

from lintrans.matrices.utility import is_valid_float, round_float

valid_floats: List[str] = [
    '0', '1', '3', '-2', '123', '-208', '1.2', '-3.5', '4.252634', '-42362.352325',
    '1e4', '-2.59e3', '4.13e-6', '-5.5244e-12'
]

invalid_floats: List[str] = [
    '', 'pi', 'e', '1.2.3', '1,2', '-', '.', 'None', 'no', 'yes', 'float'
]


@pytest.mark.parametrize('inputs,output', [(valid_floats, True), (invalid_floats, False)])
def test_is_valid_float(inputs: List[str], output: bool) -> None:
    """Test the is_valid_float() function."""
    for inp in inputs:
        assert is_valid_float(inp) == output


```

```
33      def test_round_float() -> None:
34          """Test the round_float() function."""
35          expected_values: List[Tuple[float, int, str]] = [
36              (1.0, 4, '1'), (1e-6, 4, '0'), (1e-5, 6, '1e-5'), (6.3e-8, 5, '0'), (3.2e-8, 10, '3.2e-8'),
37              (np.sqrt(2) / 2, 5, '0.70711'), (-1 * np.sqrt(2) / 2, 5, '-0.70711'),
38              (np.pi, 1, '3.1'), (np.pi, 2, '3.14'), (np.pi, 3, '3.142'), (np.pi, 4, '3.1416'), (np.pi, 5, '3.14159'),
39              (1.23456789, 2, '1.23'), (1.23456789, 3, '1.235'), (1.23456789, 4, '1.2346'), (1.23456789, 5, '1.23457'),
40              (12345.678, 1, '12345.7'), (12345.678, 2, '12345.68'), (12345.678, 3, '12345.678'),
41          ]
42
43          for num, precision, answer in expected_values:
44              assert round_float(num, precision) == answer
```

B.10 backend/matrices/utility/test_rotation_matrices.py

```
1       # lintrans - The linear transformation visualizer
2       # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4       # This program is licensed under GNU GPLv3, available here:
5       # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7       """Test functions for rotation matrices."""
8
9       from typing import List, Tuple
10
11      import numpy as np
12      import pytest
13
14      from lintrans.matrices import create_rotation_matrix
15      from lintrans.typing_ import MatrixType
16
17      angles_and_matrices: List[Tuple[float, float, MatrixType]] = [
18          (0, 0, np.array([[1, 0], [0, 1]])),
19          (90, np.pi / 2, np.array([[0, -1], [1, 0]])),
20          (180, np.pi, np.array([[-1, 0], [0, -1]])),
21          (270, 3 * np.pi / 2, np.array([[0, 1], [-1, 0]])),
22          (360, 2 * np.pi, np.array([[1, 0], [0, 1]])),
23
24          (45, np.pi / 4, np.array([
25              [np.sqrt(2) / 2, -1 * np.sqrt(2) / 2],
26              [np.sqrt(2) / 2, np.sqrt(2) / 2]
27          ])),
28          (135, 3 * np.pi / 4, np.array([
29              [-1 * np.sqrt(2) / 2, -1 * np.sqrt(2) / 2],
30              [np.sqrt(2) / 2, -1 * np.sqrt(2) / 2]
31          ])),
32          (225, 5 * np.pi / 4, np.array([
33              [-1 * np.sqrt(2) / 2, np.sqrt(2) / 2],
34              [-1 * np.sqrt(2) / 2, -1 * np.sqrt(2) / 2]
35          ])),
36          (315, 7 * np.pi / 4, np.array([
37              [np.sqrt(2) / 2, np.sqrt(2) / 2],
38              [-1 * np.sqrt(2) / 2, np.sqrt(2) / 2]
39          ])),
40
41          (30, np.pi / 6, np.array([
42              [np.sqrt(3) / 2, -1 / 2],
43              [1 / 2, np.sqrt(3) / 2]
44          ])),
45          (60, np.pi / 3, np.array([
46              [1 / 2, -1 * np.sqrt(3) / 2],
47              [np.sqrt(3) / 2, 1 / 2]
48          ])),
49          (120, 2 * np.pi / 3, np.array([
50              [-1 / 2, -1 * np.sqrt(3) / 2],
51              [np.sqrt(3) / 2, -1 / 2]
52          ])),
53          (150, 5 * np.pi / 6, np.array([
54              [-1 * np.sqrt(3) / 2, -1 / 2],
55              [1 / 2, -1 * np.sqrt(3) / 2]
```

```
        ])),
        (210, 7 * np.pi / 6, np.array([
            [-1 * np.sqrt(3) / 2, 1 / 2],
            [-1 / 2, -1 * np.sqrt(3) / 2]
        ])),
        (240, 4 * np.pi / 3, np.array([
            [-1 / 2, np.sqrt(3) / 2],
            [-1 * np.sqrt(3) / 2, -1 / 2]
        ])),
        (300, 10 * np.pi / 6, np.array([
            [1 / 2, np.sqrt(3) / 2],
            [-1 * np.sqrt(3) / 2, 1 / 2]
        ])),
        (330, 11 * np.pi / 6, np.array([
            [np.sqrt(3) / 2, 1 / 2],
            [-1 / 2, np.sqrt(3) / 2]
        ]))
    ]


    def test_create_rotation_matrix() -> None:
        """Test that create_rotation_matrix() works with given angles and expected matrices."""
        for degrees, radians, matrix in angles_and_matrices:
            assert create_rotation_matrix(degrees, degrees=True) == pytest.approx(matrix)
            assert create_rotation_matrix(radians, degrees=False) == pytest.approx(matrix)

            assert create_rotation_matrix(-1 * degrees, degrees=True) == pytest.approx(np.linalg.inv(matrix))
            assert create_rotation_matrix(-1 * radians, degrees=False) == pytest.approx(np.linalg.inv(matrix))

        assert (create_rotation_matrix(-90, degrees=True) ==
                create_rotation_matrix(270, degrees=True)).all()
        assert (create_rotation_matrix(-0.5 * np.pi, degrees=False) ==
                create_rotation_matrix(1.5 * np.pi, degrees=False)).all()
```