`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 $\mathbf{A}$ to the current scene, pause, and then apply matrix $\mathbf{B}$ - would be beneficial. This would be a good demonstration of matrix multiplication being non-commutative. $\mathbf{AB}$ is not always equal to $\mathbf{BA}$. 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.

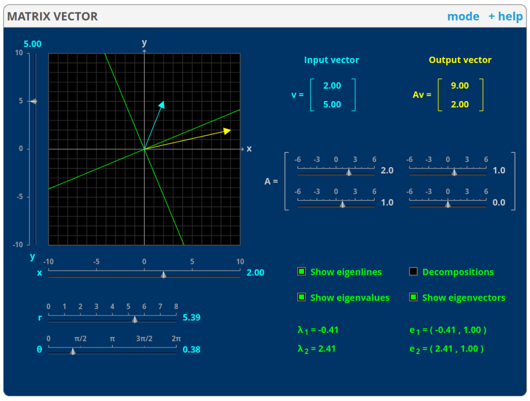


Figure 1.1: The MIT 'Matrix Vector' Mathlet

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

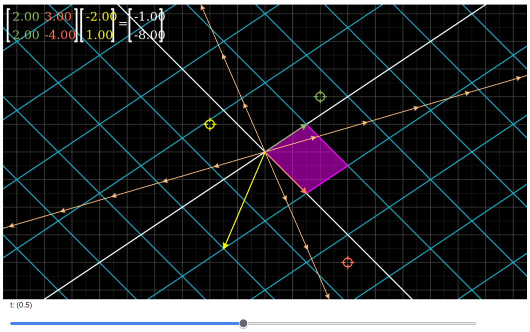


Figure 1.2: 'Linear Transformation Visualizer' halfway through an animation

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.

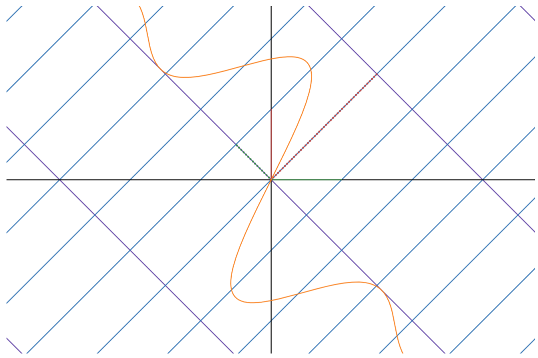


Figure 1.3: The Desmos app halfway through an animation, rendering $f(x) = \frac{\sin^2 x}{x}$ in orange

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

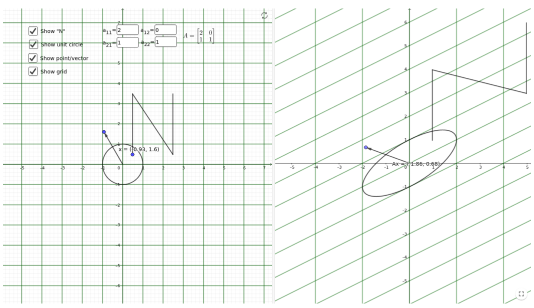


Figure 1.4: The GeoGebra applet rendering its default matrix

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 $\mathbf{C}$ is the matrix for the currently displayed transformation, and $\mathbf{T}$ is the matrix for the target transformation, then we could either animate from $\mathbf{C}$ to $\mathbf{T}$ or we could animate from $\mathbf{C}$ to $\mathbf{TC}$. 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 $\mathbf{I}$ 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 3.10 and 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.

Python 3.10 won't actually be required for the end user, because I will be compiling the app into a

[1] Python 3.10 won't compile on any earlier versions of macOS[16]

[2] Specifying a Linux version is practically impossible. Python 3.10 isn't available in many package repositories, but 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]

stand-alone 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 3.10 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. `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; 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[3]; `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 $\mathbf{C}$ to $\mathbf{TC}$, 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 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.

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

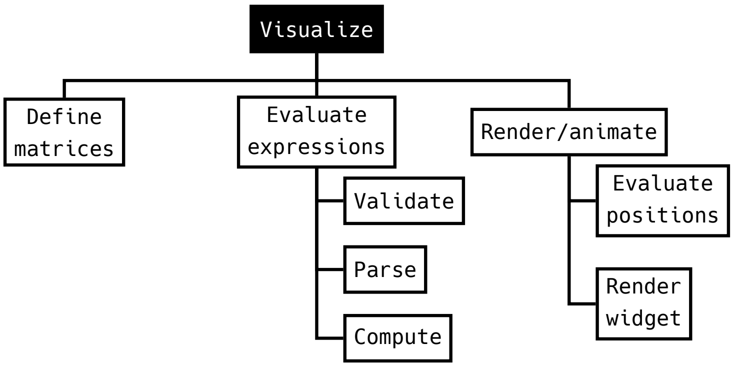
Transforming polygons is the lowest priority item on this list and will likely be implemented last, but it would definitely benefit learning. I wouldn't consider it necessary for success, but it would be very good to include, and it's certainly a feature that I want to have.

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.

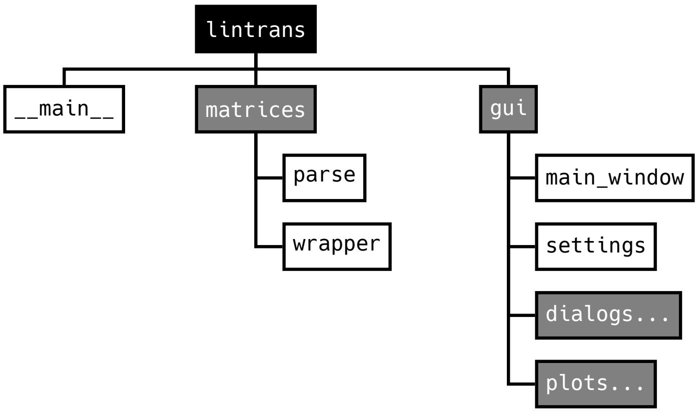
Rendering and animating will likely be the largest part in reality, but I've only decomposed it into simple blocks here. Evaluating positions involves evaluating the matrix expression that the user has input and using the columns of the resultant matrix to find the new positions of the basis vectors, 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.

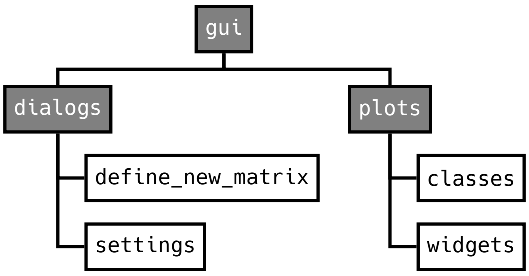
The `settings` module will contain a `DisplaySettings` dataclass[4] that will represent the settings for visualizing transformations. The `LintransMainWindow` class will have an instance of this class and check against it when rendering things. The user will be able to open a dialog to change these display 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[5].

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

[5] 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[6].

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[7] 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[8]. 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

[6] 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

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

[8] 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[9]. 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.

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

"""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"')
```

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

```

        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`.

```python
# 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
```

```python
        wrapper['Th15 Is an 1nV@l1D n@m3'] = test_matrix
        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.

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

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

        :param str name: The name of the matrix to set the value of
        :param MatrixType new_matrix: The value of the new matrix
        :rtype: None

        :raises NameError: If the name isn't a valid matrix name or is 'I'
        """
        if name not in self._matrices.keys():
            raise NameError('Matrix name must be a single capital letter')

        if name == 'I':
            raise NameError('Matrix name cannot be "I"')

        # All matrices must have float entries
        a = float(new_matrix[0][0])
        b = float(new_matrix[0][1])
        c = float(new_matrix[1][0])
        d = float(new_matrix[1][1])

        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.

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

    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 followed with a 't', then we will return the transpose of the named matrix.
        If the name is 'rot()', with a given angle in degrees, then we return a new rotation matrix with that angle.

        :param str name: The name of the matrix to get
        :returns: The value of the matrix (may be none)
        :rtype: Optional[MatrixType]

        :raises NameError: If there is no matrix with the given name
        """
        # Return a new rotation matrix
```

```python
        match = re.match(r'rot\((\d+)\)', name)
        if match is not None:
            return create_rotation_matrix(float(match.group(1)))

        # Return the transpose of this matrix
        match = re.match(r'([A-Z])t', name)
        if match is not None:
            matrix = self[match.group(1)]

            if matrix is not None:
                return matrix.T
            else:
                return None

        if name not in self._matrices:
            raise NameError(f'Unrecognised matrix name "{name}"')

        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:

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

def create_rotation_matrix(angle: float) -> MatrixType:
    """Create a matrix representing a rotation by the given number of degrees anticlockwise.

    :param float angle: The number of degrees to rotate by
    :returns MatrixType: The resultant rotation matrix
    """
    rad = np.deg2rad(angle)
    return np.array([
        [np.cos(rad), -1 * np.sin(rad)],
        [np.sin(rad), np.cos(rad)]
    ])
```

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.

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

    def parse_expression(self, expression: str) -> MatrixType:
        """Parse a given expression and return the matrix for that expression.

        Expressions are written with standard LaTeX notation for exponents. All whitespace is ignored.

        Here is documentation on syntax:
            A single matrix is written as 'A'.
            Matrix A multiplied by matrix B is written as 'AB'
            Matrix A plus matrix B is written as 'A+B'
            Matrix A minus matrix B is written as 'A-B'
            Matrix A squared is written as 'A^2'
            Matrix A to the power of 10 is written as 'A^10' or 'A^{10}'
            The inverse of matrix A is written as 'A^-1' or 'A^{-1}'
            The transpose of matrix A is written as 'A^T' or 'At'

        :param str expression: The expression to be parsed
        :returns MatrixType: The matrix result of the expression

        :raises ValueError: If the expression is invalid, such as an empty string
        """
```

```
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()
```

```python
    assert (wrapper.parse_expression('GI') == wrapper['G']).all()
    assert (wrapper.parse_expression('IG') == wrapper['G']).all()

    assert (wrapper.parse_expression('EID') == wrapper['E'] @ wrapper['D']).all()
    assert (wrapper.parse_expression('IED') == wrapper['E'] @ wrapper['D']).all()
    assert (wrapper.parse_expression('EDI') == wrapper['E'] @ wrapper['D']).all()
    assert (wrapper.parse_expression('IEIDI') == wrapper['E'] @ wrapper['D']).all()
    assert (wrapper.parse_expression('EI^3D') == wrapper['E'] @ wrapper['D']).all()


def test_simple_three_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('ABC') == wrapper['A'] @ wrapper['B'] @ wrapper['C']).all()
    assert (wrapper.parse_expression('ACB') == wrapper['A'] @ wrapper['C'] @ wrapper['B']).all()
    assert (wrapper.parse_expression('BAC') == wrapper['B'] @ wrapper['A'] @ wrapper['C']).all()
    assert (wrapper.parse_expression('EFG') == wrapper['E'] @ wrapper['F'] @ wrapper['G']).all()
    assert (wrapper.parse_expression('DAC') == wrapper['D'] @ wrapper['A'] @ wrapper['C']).all()
    assert (wrapper.parse_expression('GAE') == wrapper['G'] @ wrapper['A'] @ wrapper['E']).all()
    assert (wrapper.parse_expression('FAG') == wrapper['F'] @ wrapper['A'] @ wrapper['G']).all()
    assert (wrapper.parse_expression('GAF') == wrapper['G'] @ wrapper['A'] @ wrapper['F']).all()


def test_matrix_inverses(wrapper: MatrixWrapper) -> None:
    """Test the inverses of single 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('A^{-1}') == la.inv(wrapper['A'])).all()
    assert (wrapper.parse_expression('B^{-1}') == la.inv(wrapper['B'])).all()
    assert (wrapper.parse_expression('C^{-1}') == la.inv(wrapper['C'])).all()
    assert (wrapper.parse_expression('D^{-1}') == la.inv(wrapper['D'])).all()
    assert (wrapper.parse_expression('E^{-1}') == la.inv(wrapper['E'])).all()
    assert (wrapper.parse_expression('F^{-1}') == la.inv(wrapper['F'])).all()
    assert (wrapper.parse_expression('G^{-1}') == la.inv(wrapper['G'])).all()

    assert (wrapper.parse_expression('A^-1') == la.inv(wrapper['A'])).all()
    assert (wrapper.parse_expression('B^-1') == la.inv(wrapper['B'])).all()
    assert (wrapper.parse_expression('C^-1') == la.inv(wrapper['C'])).all()
    assert (wrapper.parse_expression('D^-1') == la.inv(wrapper['D'])).all()
    assert (wrapper.parse_expression('E^-1') == la.inv(wrapper['E'])).all()
    assert (wrapper.parse_expression('F^-1') == la.inv(wrapper['F'])).all()
    assert (wrapper.parse_expression('G^-1') == la.inv(wrapper['G'])).all()


def test_matrix_powers(wrapper: MatrixWrapper) -> None:
    """Test that matrices can be raised to integer powers."""
    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^2') == la.matrix_power(wrapper['A'], 2)).all()
    assert (wrapper.parse_expression('B^4') == la.matrix_power(wrapper['B'], 4)).all()
    assert (wrapper.parse_expression('C^{12}') == la.matrix_power(wrapper['C'], 12)).all()
    assert (wrapper.parse_expression('D^12') == la.matrix_power(wrapper['D'], 12)).all()
    assert (wrapper.parse_expression('E^8') == la.matrix_power(wrapper['E'], 8)).all()
    assert (wrapper.parse_expression('F^{-6}') == la.matrix_power(wrapper['F'], -6)).all()
    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
```

```python
    def is_valid_expression(self, expression: str) -> bool:
        """Check if the given expression is valid, using the context of the wrapper.

        This method calls _parse.validate_matrix_expression(), but also ensures
        that all the matrices in the expression are defined in the wrapper.

        :param str expression: The expression to validate
        :returns bool: Whether the expression is valid according the schema
        """
        # Get rid of the transposes to check all capital letters
        expression = re.sub(r'\^T', 't', expression)
        expression = re.sub(r'\^{T}', 't', expression)

        # Make sure all the referenced matrices are defined
        for matrix in {x for x in expression if re.match('[A-Z]', x)}:
            if self[matrix] is None:
                return False

        return _parse.validate_matrix_expression(expression)
```

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

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

"""Test the _parse.matrices module validation and parsing."""

import pytest
from lintrans._parse import validate_matrix_expression

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}',

    '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'

    '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',
]

invalid_inputs: list[str] = [
    '', 'rot()', 'A^', 'A^1.2', 'A^{3.4}', '1,2A', 'ro(12)', '5', '12^2',
    '^T', '^{12}', 'A^{13', 'A^3}', 'A^A', '^2', 'A--B', '--A'

    '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
```

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:

```python
# 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.

```python
# fd80d8d3b0e975e92dcc7c10f1f0f1276879f408
# 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+'
```

```python
    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)
```

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

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 );
```

to the new version:

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

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 );
```

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

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

def parse_matrix_expression(expression: str) -> MatrixParseList:
    """Parse the matrix expression and return a list of results.

    The return value is a list of results. This results list contains lists of tuples.
    The top list is the expressions that should be added together, and each sublist
    is expressions that should be multiplied together. These expressions to be
    multiplied are tuples, where each tuple is (multiplier, matrix identifier, index).
    The multiplier can be any real number, the matrix identifier is either a named
    matrix or a new rotation matrix declared with 'rot()', and the index is an
    integer or 'T' for transpose.

```

```python
    :param str expression: The expression to be parsed
    :returns MatrixParseTuple: A list of results
    """
    # Remove all whitespace
    expression = re.sub(r'\s', '', expression)

    # Check if it's valid
    if not validate_matrix_expression(expression):
        raise MatrixParseError('Invalid expression')

    # Wrap all exponents and transposition powers with {}
    expression = re.sub(r'(?<=\^)(-?\d+|T)(?=[^}]|$)', r'{\g<0>}', expression)

    # Remove any standalone minuses
    expression = re.sub(r'-(?=[A-Z])', '-1', expression)

    # Replace subtractions with additions
    expression = re.sub(r'-(?=\d+\.?\d*([A-Z]|rot))', '+-', expression)

    # Get rid of a potential leading + introduced by the last step
    expression = re.sub(r'^\+', '', expression)

    return [
        [
            # The tuple returned by re.findall is (multiplier, matrix identifier, full index, stripped index),
            # so we have to remove the full index, which contains the {}
            (t[0], t[1], t[3])
            for t in re.findall(r'(-?\d+\.?\d*)?([A-Z]|rot\(-?\d+\.?\d*\))(\^{(-?\d+|T)})?', group)
        ]
        # We just split the expression by '+' to have separate groups
        for group in expression.split('+')
    ]
```

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):

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

        if not self.is_valid_expression(expression):
            raise ValueError('The expression is invalid')

        parsed_result = _parse.parse_matrix_expression(expression)
        final_groups: list[list[MatrixType]] = []

        for group in parsed_result:
            f_group: list[MatrixType] = []

            for matrix in group:
                if matrix[2] == 'T':
                    m = self[matrix[1]]
                    assert m is not None
                    matrix_value = m.T
                else:
                    matrix_value = np.linalg.matrix_power(self[matrix[1]],
                                                          1 if (index := matrix[2]) == '' else int(index))

                matrix_value *= 1 if (multiplier := matrix[0]) == '' else float(multiplier)
                f_group.append(matrix_value)

            final_groups.append(f_group)

        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`.

```python
# 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

```

```
        self.label_define_new_matrix = QtWidgets.QLabel(self)
        self.label_define_new_matrix.setText('Define a\nnew matrix')
        self.label_define_new_matrix.setAlignment(QtCore.Qt.AlignCenter)

        # TODO: Implement defining a new matrix visually, numerically, as a rotation, and as an expression

        self.button_define_visually = QtWidgets.QPushButton(self)
        self.button_define_visually.setText('Visually')
        self.button_define_visually.setToolTip('Drag the basis vectors')

        self.button_define_numerically = QtWidgets.QPushButton(self)
        self.button_define_numerically.setText('Numerically')
        self.button_define_numerically.setToolTip('Define a matrix just with numbers')

        self.button_define_as_rotation = QtWidgets.QPushButton(self)
        self.button_define_as_rotation.setText('As a rotation')
        self.button_define_as_rotation.setToolTip('Define an angle to rotate by')

        self.button_define_as_expression = QtWidgets.QPushButton(self)
        self.button_define_as_expression.setText('As an expression')
        self.button_define_as_expression.setToolTip('Define a matrix in terms of other matrices')

        # Render buttons

        self.button_render = QtWidgets.QPushButton(self)
        self.button_render.setText('Render')
        self.button_render.setEnabled(False)
        self.button_render.clicked.connect(self.render_expression)
        self.button_render.setToolTip('Render the expression<br><b>(Ctrl + Enter)</b>')

        self.button_render_shortcut = QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Return'), self)
        self.button_render_shortcut.activated.connect(self.button_render.click)

        self.button_animate = QtWidgets.QPushButton(self)
        self.button_animate.setText('Animate')
        self.button_animate.setEnabled(False)
        self.button_animate.clicked.connect(self.animate_expression)
        self.button_animate.setToolTip('Animate the expression<br><b>(Ctrl + Shift + Enter)</b>')

        self.button_animate_shortcut = QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Shift+Return'), self)
        self.button_animate_shortcut.activated.connect(self.button_animate.click)

        # === Arrange widgets

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

        self.vlay_left = QVBoxLayout()
        self.vlay_left.addWidget(self.plot)
        self.vlay_left.addWidget(self.text_input_expression)

        self.vlay_misc_buttons = QVBoxLayout()
        self.vlay_misc_buttons.setSpacing(20)
        self.vlay_misc_buttons.addWidget(self.button_create_polygon)
        self.vlay_misc_buttons.addWidget(self.button_change_display_settings)

        self.vlay_define_new_matrix = QVBoxLayout()
        self.vlay_define_new_matrix.setSpacing(20)
        self.vlay_define_new_matrix.addWidget(self.label_define_new_matrix)
        self.vlay_define_new_matrix.addWidget(self.button_define_visually)
        self.vlay_define_new_matrix.addWidget(self.button_define_numerically)
        self.vlay_define_new_matrix.addWidget(self.button_define_as_rotation)
        self.vlay_define_new_matrix.addWidget(self.button_define_as_expression)

        self.vlay_render = QVBoxLayout()
        self.vlay_render.setSpacing(20)
        self.vlay_render.addWidget(self.button_animate)
        self.vlay_render.addWidget(self.button_render)

        self.vlay_right = QVBoxLayout()
        self.vlay_right.setSpacing(50)
        self.vlay_right.addLayout(self.vlay_misc_buttons)
        self.vlay_right.addLayout(self.vlay_define_new_matrix)
        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()
```

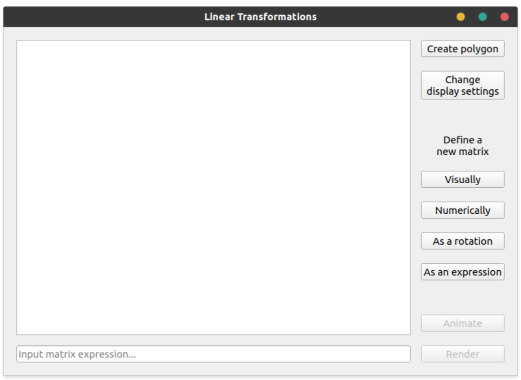


Figure 3.1: The first version of the GUI

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

"""The module to provide dialogs for defining new matrices."""

from numpy import array
from PyQt5 import QtGui, QtWidgets
from PyQt5.QtWidgets import QDialog, QGridLayout, QHBoxLayout, QVBoxLayout
```

```python

from lintrans.matrices import MatrixWrapper

ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'


def is_float(string: str) -> bool:
    """Check if a string is a float."""
    try:
        float(string)
        return True
    except ValueError:
        return False


class DefineNumericallyDialog(QDialog):
    """The dialog class that allows the user to define a new matrix numerically."""

    def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
        """Create the dialog, but don't run it yet.

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

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

        # === Create the widgets

        self.button_confirm = QtWidgets.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>')

        QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Return'), self).activated.connect(self.button_confirm.click)

        self.button_cancel = QtWidgets.QPushButton(self)
        self.button_cancel.setText('Cancel')
        self.button_cancel.clicked.connect(self.close)
        self.button_cancel.setToolTip('Cancel this definition<br><b>(Ctrl + Q)</b>')

        QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Q'), self).activated.connect(self.button_cancel.click)

        self.element_tl = QtWidgets.QLineEdit(self)
        self.element_tl.textChanged.connect(self.update_confirm_button)

        self.element_tr = QtWidgets.QLineEdit(self)
        self.element_tr.textChanged.connect(self.update_confirm_button)

        self.element_bl = QtWidgets.QLineEdit(self)
        self.element_bl.textChanged.connect(self.update_confirm_button)

        self.element_br = QtWidgets.QLineEdit(self)
        self.element_br.textChanged.connect(self.update_confirm_button)

        self.matrix_elements = (self.element_tl, self.element_tr, self.element_bl, self.element_br)

        self.letter_combo_box = QtWidgets.QComboBox(self)

        # Everything except I, because that's the identity
        for letter in ALPHABET_NO_I:
            self.letter_combo_box.addItem(letter)

        self.letter_combo_box.activated.connect(self.load_matrix)

        # === Arrange the widgets

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

        self.grid_matrix = QGridLayout()
```

```python
        self.grid_matrix.setSpacing(20)
        self.grid_matrix.addWidget(self.element_tl, 0, 0)
        self.grid_matrix.addWidget(self.element_tr, 0, 1)
        self.grid_matrix.addWidget(self.element_bl, 1, 0)
        self.grid_matrix.addWidget(self.element_br, 1, 1)

        self.hlay_buttons = QHBoxLayout()
        self.hlay_buttons.setSpacing(20)
        self.hlay_buttons.addWidget(self.button_cancel)
        self.hlay_buttons.addWidget(self.button_confirm)

        self.vlay_right = QVBoxLayout()
        self.vlay_right.setSpacing(20)
        self.vlay_right.addLayout(self.grid_matrix)
        self.vlay_right.addLayout(self.hlay_buttons)

        self.hlay_all = QHBoxLayout()
        self.hlay_all.setSpacing(20)
        self.hlay_all.addWidget(self.letter_combo_box)
        self.hlay_all.addLayout(self.vlay_right)

        self.setLayout(self.hlay_all)

        # Finally, we load the default matrix A into the boxes
        self.load_matrix(0)

    def update_confirm_button(self) -> None:
        """Enable the confirm button if there are numbers in every box."""
        for elem in self.matrix_elements:
            if elem.text() == '' or not is_float(elem.text()):
                # If they're not all numbers, then we can't confirm it
                self.button_confirm.setEnabled(False)
                return

        # If we didn't find anything invalid
        self.button_confirm.setEnabled(True)

    def load_matrix(self, index: int) -> None:
        """If the selected matrix is defined, load it into the boxes."""
        matrix = self.matrix_wrapper[ALPHABET_NO_I[index]]

        if matrix is None:
            for elem in self.matrix_elements:
                elem.setText('')

        else:
            self.element_tl.setText(str(matrix[0][0]))
            self.element_tr.setText(str(matrix[0][1]))
            self.element_bl.setText(str(matrix[1][0]))
            self.element_br.setText(str(matrix[1][1]))

        self.update_confirm_button()

    def confirm_matrix(self) -> None:
        """Confirm the inputted matrix and assign it to the name."""
        letter = self.letter_combo_box.currentText()
        matrix = array([
            [float(self.element_tl.text()), float(self.element_tr.text())],
            [float(self.element_bl.text()), float(self.element_br.text())]
        ])

        self.matrix_wrapper[letter] = matrix
        self.close()
```

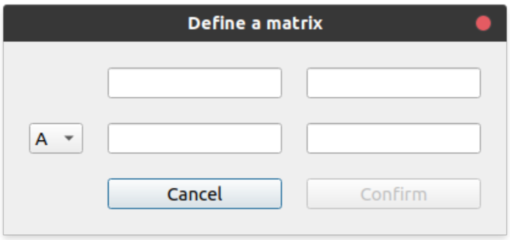


Figure 3.2: The first version of the numerical definition dialog

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

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

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

class DefineDialog(QDialog):
    """A superclass for definitions dialogs."""

    def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
        """Create the dialog, but don't run it yet.

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

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

        # === Create the widgets

        self.button_confirm = QtWidgets.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)

        self.button_cancel = QtWidgets.QPushButton(self)
        self.button_cancel.setText('Cancel')
        self.button_cancel.clicked.connect(self.close)
        self.button_cancel.setToolTip('Cancel this definition<br><b>(Ctrl + Q)</b>')
        QShortcut(QKeySequence('Ctrl+Q'), self).activated.connect(self.button_cancel.click)

        self.label_equals = QtWidgets.QLabel()
```

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

        self.letter_combo_box = QtWidgets.QComboBox(self)

        # Everything except I, because that's the identity
        for letter in ALPHABET_NO_I:
            self.letter_combo_box.addItem(letter)

        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

        # === Arrange the widgets

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

        self.horizontal_spacer = QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum)

        self.hlay_buttons = QHBoxLayout()
        self.hlay_buttons.setSpacing(20)
        self.hlay_buttons.addItem(self.horizontal_spacer)
        self.hlay_buttons.addWidget(self.button_cancel)
        self.hlay_buttons.addWidget(self.button_confirm)

    @property
    def selected_letter(self) -> str:
        """The letter currently selected in the combo box."""
        return self.letter_combo_box.currentText()

    @abc.abstractmethod
    def update_confirm_button(self) -> None:
        """Enable the confirm button if it should be enabled."""
        ...

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

        This should mutate self.matrix_wrapper and then call self.accept().
        """
        ...
```

I then added the class for the rotation definition dialog.

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

class DefineAsARotationDialog(DefineDialog):
    """The dialog that allows the user to define a new matrix as a rotation."""

    def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
        """Create the dialog, but don't run it yet."""
        super().__init__(matrix_wrapper, *args, **kwargs)

        # === Create the widgets

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

        self.text_angle = QtWidgets.QLineEdit(self)
        self.text_angle.setPlaceholderText('angle')
        self.text_angle.textChanged.connect(self.update_confirm_button)

        self.label_close_paren = QtWidgets.QLabel(self)
```

```python
        self.label_close_paren.setText(')')

        self.checkbox_radians = QtWidgets.QCheckBox(self)
        self.checkbox_radians.setText('Radians')

        # === Arrange the widgets

        self.hlay_checkbox_and_buttons = QHBoxLayout()
        self.hlay_checkbox_and_buttons.setSpacing(20)
        self.hlay_checkbox_and_buttons.addWidget(self.checkbox_radians)
        self.hlay_checkbox_and_buttons.addItem(self.horizontal_spacer)
        self.hlay_checkbox_and_buttons.addLayout(self.hlay_buttons)

        self.hlay_definition = QHBoxLayout()
        self.hlay_definition.addWidget(self.letter_combo_box)
        self.hlay_definition.addWidget(self.label_equals)
        self.hlay_definition.addWidget(self.text_angle)
        self.hlay_definition.addWidget(self.label_close_paren)

        self.vlay_all = QVBoxLayout()
        self.vlay_all.setSpacing(20)
        self.vlay_all.addLayout(self.hlay_definition)
        self.vlay_all.addLayout(self.hlay_checkbox_and_buttons)

        self.setLayout(self.vlay_all)

    def update_confirm_button(self) -> None:
        """Enable the confirm button if there is a valid float in the angle box."""
        self.button_confirm.setEnabled(is_float(self.text_angle.text()))

    def confirm_matrix(self) -> None:
        """Confirm the inputted matrix and assign it."""
        self.matrix_wrapper[self.selected_letter] = create_rotation_matrix(
            float(self.text_angle.text()),
            degrees=not self.checkbox_radians.isChecked()
        )
        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.

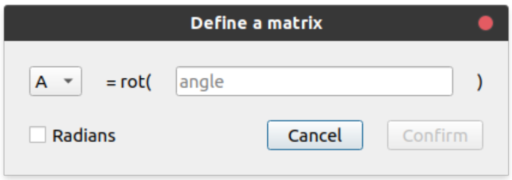


Figure 3.3: The first version of the rotation definition dialog

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.

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

    def dialog_define_matrix(self, dialog_class: Type[DefineDialog]) -> None:
        """Open a generic definition dialog to define a new matrix.

        The class for the desired dialog is passed as an argument. We create an
        instance of this class and the dialog is opened asynchronously and modally
        (meaning it blocks interaction with the main window) with the proper method
        connected to the ``dialog.finished`` slot.

        .. note::
```

```python
            ``dialog_class`` must subclass :class:`lintrans.gui.dialogs.define_new_matrix.DefineDialog`.

        :param dialog_class: The dialog class to instantiate
        :type dialog_class: Type[lintrans.gui.dialogs.define_new_matrix.DefineDialog]
        """
        # We create a dialog with a deepcopy of the current matrix_wrapper
        # This avoids the dialog mutating this one
        dialog = dialog_class(deepcopy(self.matrix_wrapper), self)

        # .open() is asynchronous and doesn't spawn a new event loop, but the dialog is still modal (blocking)
        dialog.open()

        # So we have to use the finished slot to call a method when the user accepts the dialog
        # If the user rejects the dialog, this matrix_wrapper will be the same as the current one, because we copied it
        # So we don't care, we just assign the wrapper anyway
        dialog.finished.connect(lambda: self._assign_matrix_wrapper(dialog.matrix_wrapper))

    def _assign_matrix_wrapper(self, matrix_wrapper: MatrixWrapper) -> None:
        """Assign a new value to self.matrix_wrapper.

        This is a little utility function that only exists because a lambda
        callback can't directly assign a value to a class attribute.

        :param matrix_wrapper: The new value of the matrix wrapper to assign
        :type matrix_wrapper: MatrixWrapper
        """
        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.

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

class DefineAsAnExpressionDialog(DefineDialog):
    """The dialog that allows the user to define a matrix as an expression."""

    def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
        """Create the dialog, but don't run it yet."""
        super().__init__(matrix_wrapper, *args, **kwargs)

        self.setMinimumWidth(450)

        # === Create the widgets

        self.text_box_expression = QtWidgets.QLineEdit(self)
        self.text_box_expression.setPlaceholderText('Enter matrix expression...')
        self.text_box_expression.textChanged.connect(self.update_confirm_button)

        # === Arrange the widgets

        self.hlay_definition.addWidget(self.text_box_expression)

        self.vlay_all = QVBoxLayout()
        self.vlay_all.setSpacing(20)
        self.vlay_all.addLayout(self.hlay_definition)
        self.vlay_all.addLayout(self.hlay_buttons)

        self.setLayout(self.vlay_all)

    def update_confirm_button(self) -> None:
        """Enable the confirm button if the expression is valid."""
        self.button_confirm.setEnabled(
            self.matrix_wrapper.is_valid_expression(self.text_box_expression.text())
        )

    def confirm_matrix(self) -> None:
        """Evaluate the matrix expression and assign its value to the chosen matrix."""
        self.matrix_wrapper[self.selected_letter] = \
```

```
            self.matrix_wrapper.evaluate_expression(self.text_box_expression.text())
        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.

```python
# 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
```

```python
        qp.setPen(QPen(self.axes_colour, self.line_width))
        qp.drawLine(self.w // 2, 0, self.w // 2, self.h)
        qp.drawLine(0, self.h // 2, self.w, self.h // 2)


class ViewTransformationWidget(TransformationPlotWidget):
    """This class is used to visualise matrices as transformations."""

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

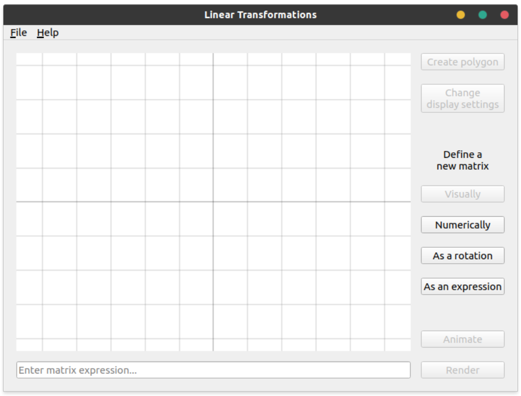


Figure 3.4: The GUI with background axes

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.

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

    @property
    def origin(self) -> tuple[int, int]:
        """Return the canvas coords of the origin."""
        return self.width() // 2, self.height() // 2

    def trans_x(self, x: float) -> int:
        """Transform an x coordinate from grid coords to canvas coords."""
        return int(self.origin[0] + x * self.grid_spacing)

    def trans_y(self, y: float) -> int:
        """Transform a y coordinate from grid coords to canvas coords."""
        return int(self.origin[1] - y * self.grid_spacing)

    def trans_coords(self, x: float, y: float) -> tuple[int, int]:
        """Transform a coordinate in grid coords to canvas coords."""
        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.

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

```python
class ViewTransformationWidget(TransformationPlotWidget):
    """This class is used to visualise matrices as transformations."""

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

        self.point_i: tuple[float, float] = (1., 0.)
        self.point_j: tuple[float, float] = (0., 1.)

        self.colour_i = QColor(37, 244, 15)
        self.colour_j = QColor(8, 8, 216)

        self.width_vector_line = 1
        self.width_transformed_grid = 0.6

    def transform_by_matrix(self, matrix: MatrixType) -> None:
        """Transform the plane by the given matrix."""
        self.point_i = (matrix[0][0], matrix[1][0])
        self.point_j = (matrix[0][1], matrix[1][1])
        self.update()
```

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

```python
# 37e7c208a33d7cbbc8e0bb6c94cd889e2918c605
# 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))
```

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

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

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

Testing this new code shows that it works well.

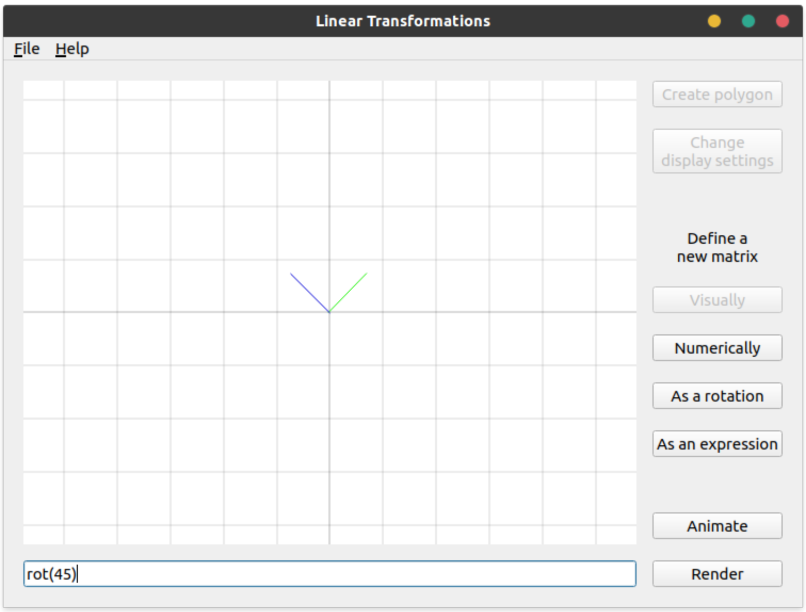


Figure 3.5: Basis vectors drawn for a 45° rotation

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

    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)
```

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

    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

        if vector_x == 0:
            painter.drawLine(self.trans_x(0), 0, self.trans_x(0), self.height())

            for i in range(int(max_x / point_x)):
                painter.drawLine(
                    self.trans_x((i + 1) * point_x),
                    0,
                    self.trans_x((i + 1) * point_x),
                    self.height()
                )
                painter.drawLine(
                    self.trans_x(-1 * (i + 1) * point_x),
```

```python
                    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()`.

```python
# 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.

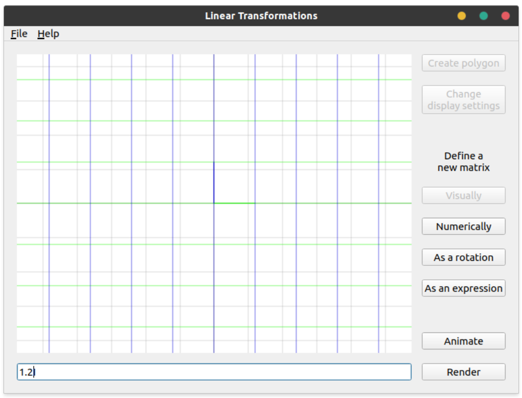


Figure 3.6: Parallel lines being drawn for matrix $1.2\mathbf{I}$

```python
# 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
```

```
        print(max_x, max_y, vector_x, vector_y, point_x, point_y)

        # We want to use y = mx + c but m = y / x and if either of those are 0, then this
        # equation is harder to work with, so we deal with these edge cases first
        if 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

        elif abs(vector_x) < 1e-12:
            painter.drawLine(self.trans_x(0), 0, self.trans_x(0), self.height())

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

        elif abs(vector_y) < 1e-12:
            painter.drawLine(0, self.trans_y(0), self.width(), self.trans_y(0))

            for i in range(abs(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)
                )

        else:  # If the line is not horizontal or vertical, then we can use y = mx + c
            m = vector_y / vector_x
            c = point_y - m * point_x

            # For c = 0
            painter.drawLine(
                *self.trans_coords(
                    -1 * max_x,
                    m * -1 * max_x
                ),
                *self.trans_coords(
                    max_x,
                    m * max_x
                )
            )

            # Count up how many multiples of c we can have without wasting time rendering lines off screen
            multiples_of_c: int = 0
            ii: int = 1
            while True:
                y1 = m * max_x + ii * c
                y2 = -1 * m * max_x + ii * c

                if y1 < max_y or y2 < max_y:
                    multiples_of_c += 1
                    ii += 1

                else:
                    break
```

```python
        # Once we know how many lines we can draw, we just draw them all
        for i in range(1, multiples_of_c + 1):
            painter.drawLine(
                *self.trans_coords(
                    -1 * max_x,
                    m * -1 * max_x + i * c
                ),
                *self.trans_coords(
                    max_x,
                    m * max_x + i * c
                )
            )
            painter.drawLine(
                *self.trans_coords(
                    -1 * max_x,
                    m * -1 * max_x - i * c
                ),
                *self.trans_coords(
                    max_x,
                    m * max_x - i * c
                )
            )
```

This code checks if x or y is zero[10] 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.

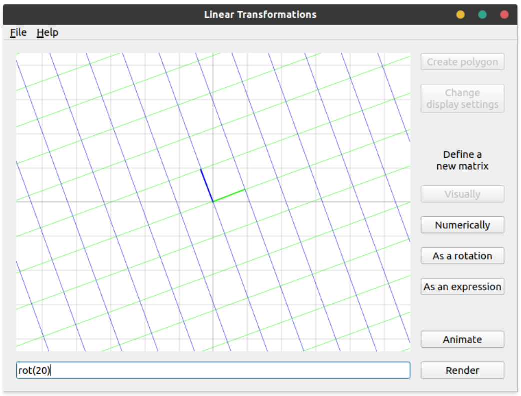


Figure 3.7: An example of a 20° rotation

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.

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

    def animate_expression(self) -> None:
        """Animate the expression in the input box, and then clear the box."""
        self.button_render.setEnabled(False)
        self.button_animate.setEnabled(False)

        matrix = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())
        matrix_move = matrix - self.matrix_wrapper['I']
        steps: int = 100

        for i in range(0, steps + 1):
```

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

```python
            self.plot.visualize_matrix_transformation(
                self.matrix_wrapper['I'] + (i / steps) * matrix_move
            )

            self.update()
            self.repaint()

            time.sleep(0.01)

        self.button_render.setEnabled(False)
        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.

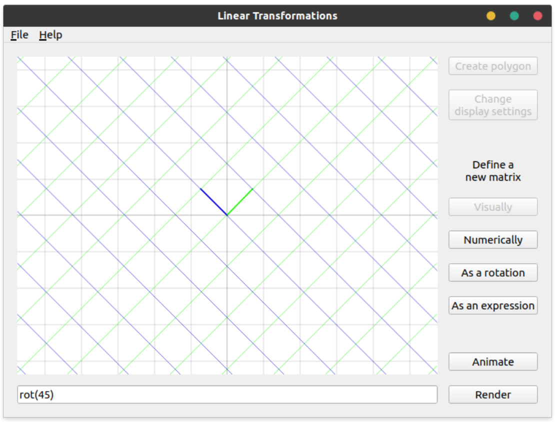


Figure 3.8: What we would expect halfway through a 90° rotation

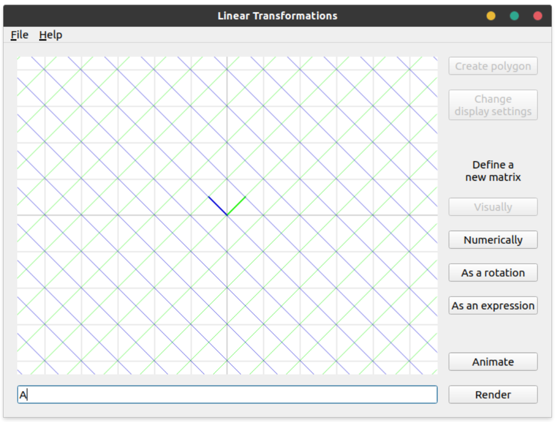


Figure 3.9: What we actually get halfway through a 90° rotation

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 $\dfrac{1}{\det(\mathbf{A})}$ 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 $c = \dfrac{1}{\sqrt{|\det(\mathbf{A})|}}$. 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:

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

        # Get the target matrix and it's determinant
        matrix_target = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())
        det_target = linalg.det(matrix_target)

        identity = self.matrix_wrapper['I']
        steps: int = 100

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

            # matrix_a is the identity 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 = identity + proportion * (matrix_target - identity)

            # So 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) = 1, so then we can scale it with the animation
            # So we get c^2 det(A) = 1 => c = sqrt(1 / abs(det(A)))
            # Then we scale A down to get a determinant of 1, and call that matrix_b
            if det_a == 0:
                c = 0
            else:
                c = np.sqrt(1 / abs(det_a))

            matrix_b = c * matrix_a

            # matrix_c is the final matrix that we transform by
            # It's B, but we scale it up 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))
            # But we defined B to have det 1, so we can ignore it there

            # 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)) - 1)

            matrix_c = scalar * matrix_b

            self.plot.visualize_matrix_transformation(matrix_c)

            self.repaint()
            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.

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

        else:  # If the line is not horizontal or vertical, then we can use y = mx + c
            m = vector_y / vector_x
            c = point_y - m * point_x

            # For c = 0
            painter.drawLine(
                *self.trans_coords(
                    -1 * max_x,
                    m * -1 * max_x
                ),
                *self.trans_coords(
                    max_x,
                    m * max_x
                )
            )

            # We keep looping and increasing the multiple of c until we stop drawing lines on the canvas
            multiple_of_c = 1
            while self.draw_pair_of_oblique_lines(painter, m, multiple_of_c * c):
                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:

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

    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 ``QPainter`` object to use for drawing the vectors and grid lines
        :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 ``QPainter`` object to use for drawing the vectors and grid lines
```

```
        :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.trans_coords(*points[0]),
                *self.trans_coords(*points[1])
            )
            return True
```

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

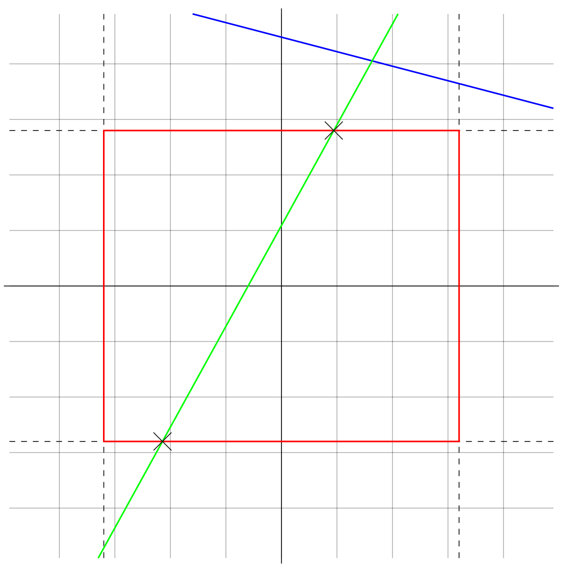


Figure 3.10: Two example lines and the viewport box

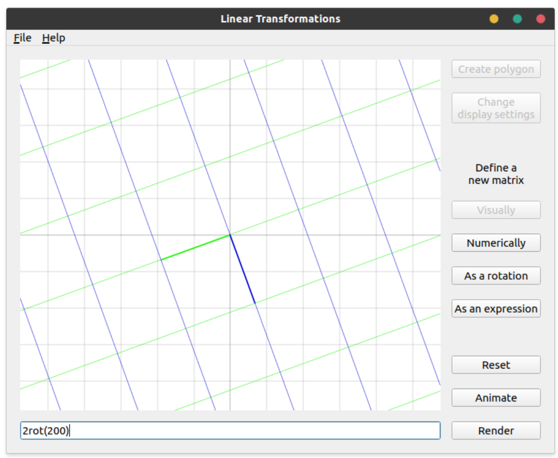


Figure 3.11: A demonstration of the new oblique lines system.

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

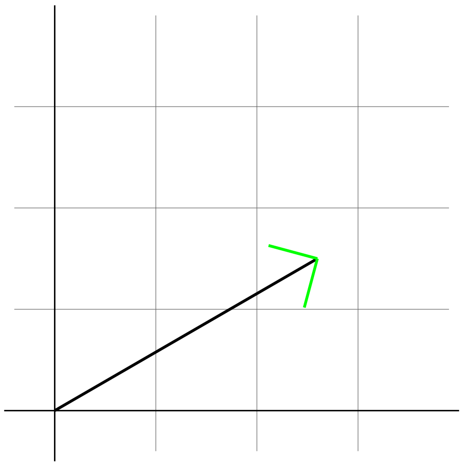


Figure 3.12: An example of a vector with the arrowheads highlighted in green

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][11], which I adapted for Python.

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

    def draw_arrowhead_away_from_origin(self, painter: QPainter, point: tuple[float, float]) -> None:
        """Draw an arrowhead at ``point``, pointing away from the origin.

        :param QPainter painter: The ``QPainter`` object to use to draw the arrowheads with
        :param point: The point to draw the arrowhead at, given in grid coords
        :type point: tuple[float, float]
        """
        # This algorithm was adapted from a C# algorithm found at
        # http://csharphelper.com/blog/2014/12/draw-lines-with-arrowheads-in-c/

        # Get the x and y coords of the point, and then normalize them
        # We have to normalize them, or else the size of the arrowhead will
        # scale with the distance of the point from the origin
        x, y = point
        nx = x / np.sqrt(x * x + y * y)
        ny = y / np.sqrt(x * x + y * y)

        # We choose a length and do some magic to find the steps in the x and y directions
        length = 0.15
        dx = length * (-nx - ny)
        dy = length * (nx - ny)

        # Then we just plot those lines
        painter.drawLine(*self.trans_coords(x, y), *self.trans_coords(x + dx, y + dy))
        painter.drawLine(*self.trans_coords(x, y), *self.trans_coords(x - dy, y + dx))
```

[11]This website is currently being rewritten and this arrowheads tutorial is listed as 'not rebuilt' on `http://csharphelper.com`

```

    def draw_vector_arrowheads(self, painter: QPainter) -> None:
        """Draw arrowheads at the tips of the basis vectors.

        :param QPainter painter: The ``QPainter`` object to use to draw the arrowheads with
        """
        painter.setPen(QPen(self.colour_i, self.width_vector_line))
        self.draw_arrowhead_away_from_origin(painter, self.point_i)
        painter.setPen(QPen(self.colour_j, self.width_vector_line))
        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()`.

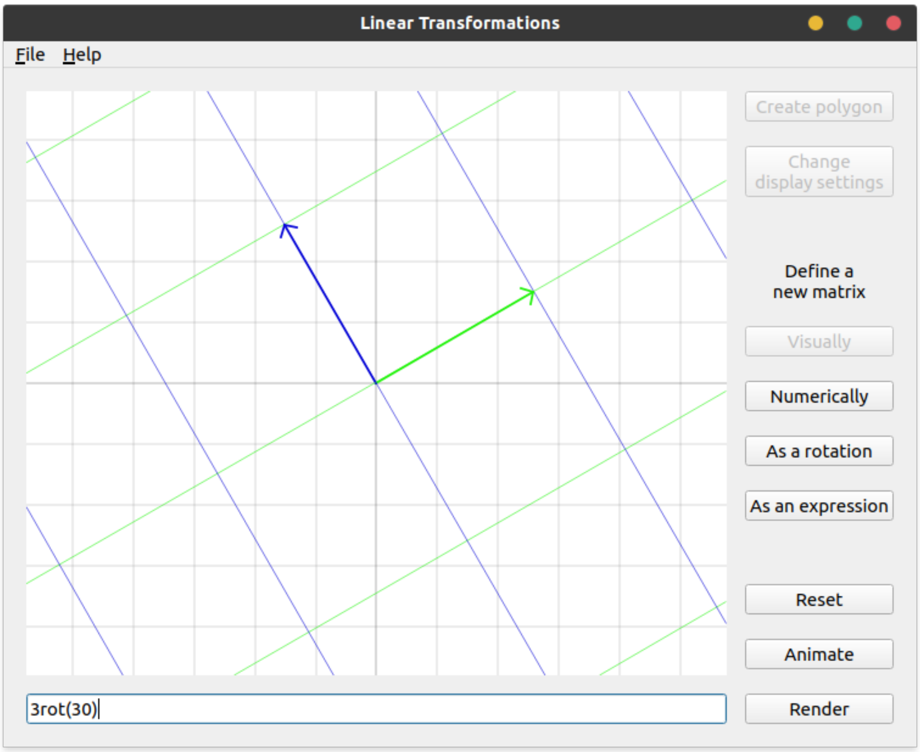


Figure 3.13: An example of the i and j vectors with arrowheads

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

    default_grid_spacing: int = 50

    def __init__(self, *args, **kwargs):
        """Create the widget and setup backend stuff for rendering.

        .. note:: ``*args`` and ``**kwargs`` are passed the superclass constructor (``QWidget``).
        """
```

```python
        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.colour_background_grid = QColor(128, 128, 128)
        self.colour_background_axes = QColor(0, 0, 0)

        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.

```python
# 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.

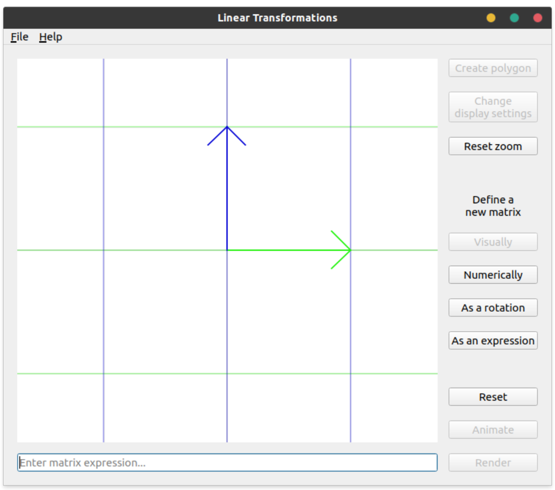


Figure 3.14: The GUI zoomed in a bit

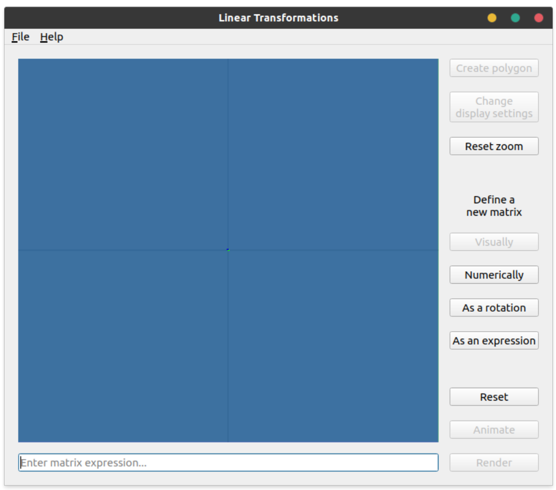


Figure 3.15: The GUI zoomed out as far as possible

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__()`.

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

        vector_length = np.sqrt(x * x + y * y)
        nx = x / vector_length
        ny = y / vector_length

        # We choose a length and find the steps in the x and y directions
        length = min(
            self.arrowhead_length * self.default_grid_spacing / self.grid_spacing,
            vector_length
        )
```

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

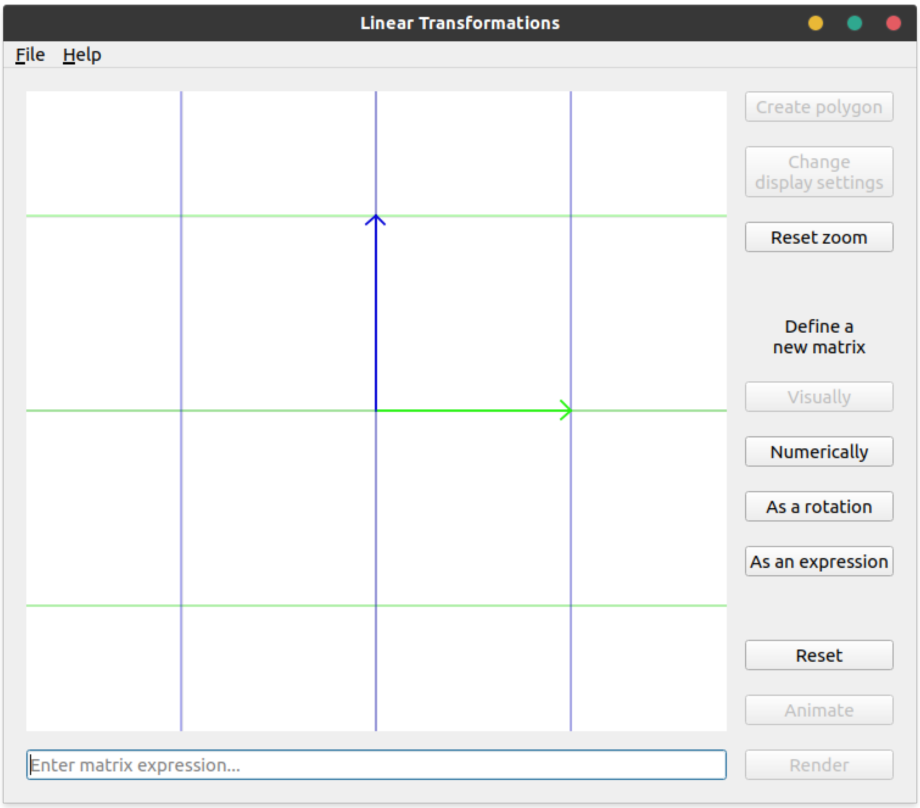


Figure 3.16: The arrowheads adjusted for zoom level

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 $\mathbf{0}$ 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.

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

        # 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:
                self.draw_oblique_line(painter, vector_y / vector_x, 0)

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

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

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

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

    def show_error_message(self, title: str, text: str, info: str | None = None) -> None:
        """Show an error message in a dialog box.

        :param str title: The window title of the dialog box
        :param str text: The simple error message
        :param info: The more informative error message
        :type info: Optional[str]
        """
        dialog = QMessageBox(self)
        dialog.setIcon(QMessageBox.Critical)
        dialog.setWindowTitle(title)
        dialog.setText(text)

        if info is not None:
            dialog.setInformativeText(info)

        dialog.open()

        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()`.

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

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

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

        Convert the elements of the matrix to canvas coords and make sure they fit within Qt's 32-bit integer limit.

        :param MatrixType matrix: The matrix to check
        :returns bool: Whether the matrix fits on the canvas
        """
        coords: list[tuple[int, int]] = [self.plot.trans_coords(*vector) for vector in matrix.T]

        for x, y in coords:
            if not (-2147483648 <= x <= 2147483647 and -2147483648 <= y <= 2147483647):
                return True

        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.

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

class DefineVisuallyDialog(DefineDialog):
    """The dialog class that allows the user to define a matrix visually."""

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

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

        self.setMinimumSize(500, 450)

        # === Create the widgets

        self.combobox_letter.activated.connect(self.show_matrix)

        self.plot = DefineVisuallyWidget(self)

        # === Arrange the widgets

        self.hlay_definition.addWidget(self.plot)
        self.hlay_definition.setStretchFactor(self.plot, 1)

        self.vlay_all = QVBoxLayout()
        self.vlay_all.setSpacing(20)
        self.vlay_all.addLayout(self.hlay_definition)
        self.vlay_all.addLayout(self.hlay_buttons)

        self.setLayout(self.vlay_all)

        # We load the default matrix A into the plot
        self.show_matrix(0)

        # We also enable the confirm button, because any visually defined matrix is valid
        self.button_confirm.setEnabled(True)

    def update_confirm_button(self) -> None:
        """Enable the confirm button.

        .. note::
           The confirm button is always enabled in this dialog and this method is never actually used,
           so it's got an empty body. It's only here because we need to implement the abstract method.
        """

```

```python
    def show_matrix(self, index: int) -> None:
        """Show the selected matrix on the plot. If the matrix is None, show the identity."""
        matrix = self.matrix_wrapper[ALPHABET_NO_I[index]]

        if matrix is None:
            matrix = self.matrix_wrapper['I']

        self.plot.visualize_matrix_transformation(matrix)
        self.plot.update()

    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.

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

    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
```

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.

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

class DefineVisuallyWidget(VisualizeTransformationWidget):
    """This class is the widget that allows the user to visually define a matrix.

    This is just the widget itself. If you want the dialog, use
    :class:`lintrans.gui.dialogs.define_new_matrix.DefineVisuallyDialog`.
    """

    def __init__(self, *args, **kwargs):
        """Create the widget and enable mouse tracking. ``*args`` and ``**kwargs`` are passed to ``super()``."""
        super().__init__(*args, **kwargs)

        # self.setMouseTracking(True)
        self.dragged_point: tuple[float, float] | None = None

        # This is the distance that the cursor needs to be from the point to drag it
        self.epsilon: int = 5

    def mousePressEvent(self, event: QMouseEvent) -> None:
        """Handle a QMouseEvent when the user pressed a button."""
        mx = event.x()
        my = event.y()
        button = event.button()

        if button != Qt.LeftButton:
```

```python
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()`.

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

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

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

3.4.9 Implementing transitional animation

Currently, all animation animates from **I** to the target matrix **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 **S** to the target matrix **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 **S** be the starting matrix, and **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 $c = \sqrt{\left|\frac{\det(\mathbf{S})}{\det(\mathbf{A})}\right|}$ 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 **T** and find that $d = \sqrt{\left|\frac{\det(\mathbf{T})}{\det(\mathbf{B})}\right|}$. 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 $s = 1 + p\left(\sqrt{\left|\frac{\det(\mathbf{T})}{\det(\mathbf{B})}\right|} - 1\right)$ and render $\mathbf{C} = s\mathbf{B}$ on each frame.

In code, that looks like this:

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

    def animate_expression(self) -> None:
        """Animate from the current matrix to the matrix in the expression box."""
        self.button_render.setEnabled(False)
        self.button_animate.setEnabled(False)

        # Get the target matrix and it's determinant
        try:
            matrix_target = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())

        except linalg.LinAlgError:
            self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
            return

        matrix_start: MatrixType = np.array([
            [self.plot.point_i[0], self.plot.point_j[0]],
            [self.plot.point_i[1], self.plot.point_j[1]]
        ])

        self.animate_between_matrices(matrix_start, matrix_target)
```

```python

        self.button_render.setEnabled(True)
        self.button_animate.setEnabled(True)

    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)

            # So 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_c 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)

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

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

            self.plot.visualize_matrix_transformation(matrix_c)

            # 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()
```

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 $\mathbf{C}$ representing the current scene to the composition $\mathbf{TC}$ with the target matrix $\mathbf{T}$.

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

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

    def animate_expression(self) -> None:
        """Animate from the current matrix to the matrix in the expression box."""
        self.button_render.setEnabled(False)
        self.button_animate.setEnabled(False)

        matrix_start: MatrixType = np.array([
            [self.plot.point_i[0], self.plot.point_j[0]],
            [self.plot.point_i[1], self.plot.point_j[1]]
        ])

        text = self.lineedit_expression_box.text()

        # If there's commas in the expression, then we want to animate each part at a time
        if ',' in text:
            current_matrix = matrix_start

            # For each expression in the list, right multiply it by the current matrix,
            # and animate from the current matrix to that new matrix
            for expr in text.split(',')[::-1]:
                new_matrix = self.matrix_wrapper.evaluate_expression(expr) @ current_matrix

                self.animate_between_matrices(current_matrix, new_matrix)
                current_matrix = new_matrix

                # Here we just redraw and allow for other events to be handled while we pause
                self.plot.update()
                QApplication.processEvents()
                QThread.msleep(500)

        # If there's no commas, then just animate directly from the start to the target
        else:
            # Get the target matrix and it's determinant
            try:
                matrix_target = self.matrix_wrapper.evaluate_expression(text)

            except linalg.LinAlgError:
                self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
                return

            self.animate_between_matrices(matrix_start, matrix_target)
```

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

```python

        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 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:

```python
# 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)
```

```python
# 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`.

```python
# 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."""

    animate_determinant: bool = True
    """This controls whether we want the determinant to change smoothly during the animation."""

    applicative_animation: bool = True
    """There are two types of simple animation, transitional and applicative.

    Let ``C`` be the matrix representing the currently displayed transformation, and let ``T`` be the target matrix.
    Transitional animation means that we animate directly from ``C`` from ``T``,
    and applicative animation means that we animate from ``C`` to ``TC``, so we apply ``T`` to ``C``.
    """

    animation_pause_length: int = 400
    """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

    def animate_expression(self) -> None:
        """Animate from the current matrix to the matrix in the expression box."""
        self.button_render.setEnabled(False)
        self.button_animate.setEnabled(False)

        matrix_start: MatrixType = np.array([
            [self.plot.point_i[0], self.plot.point_j[0]],
            [self.plot.point_i[1], self.plot.point_j[1]]
        ])

        text = self.lineedit_expression_box.text()

        # If there's commas in the expression, then we want to animate each part at a time
        if ',' in text:
            current_matrix = matrix_start

            # For each expression in the list, right multiply it by the current matrix,
            # and animate from the current matrix to that new matrix
            for expr in text.split(',')[::-1]:
                new_matrix = self.matrix_wrapper.evaluate_expression(expr) @ current_matrix

                self.animate_between_matrices(current_matrix, new_matrix)
                current_matrix = new_matrix

                # Here we just redraw and allow for other events to be handled while we pause
                self.plot.update()
                QApplication.processEvents()
                QThread.msleep(self.display_settings.animation_pause_length)

        # If there's no commas, then just animate directly from the start to the target
        else:
            # Get the target matrix and it's determinant
            try:
                matrix_target = self.matrix_wrapper.evaluate_expression(text)

            except linalg.LinAlgError:
                self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
                return

            # The concept of applicative animation is explained in /gui/settings.py
            if self.display_settings.applicative_animation:
                matrix_target = matrix_target @ matrix_start
```

```python

            self.animate_between_matrices(matrix_start, matrix_target)

        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
# 2041c7a24d963d8d142d6f0f20ec3828ba8257c6
# 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:
                # 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)

                # If we're animating towards a det 0 matrix, then we don't want to scale the
                # determinant with the animation, because this makes the process not work
                # I'm doing this here rather than wrapping the whole animation logic in an
                # if block mainly because this looks nicer than an extra level of indentation
                # The extra processing cost is negligible thanks to NumPy's optimizations
                if det_target == 0:
                    matrix_to_render = matrix_a
                else:
                    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/blog/2014/12/draw-lines-with-arrowheads-in-c/.

[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 `__init__.py`

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

from . import gui, matrices, typing_

__version__ = '0.3.0-alpha'

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

A.2 `__main__.py`

```python
#!/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."""

import sys
from argparse import ArgumentParser
from textwrap import dedent
from typing import List

from lintrans import __version__, gui


def main(args: List[str]) -> 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(
        '-h',
        '--help',
        default=False,
        action='store_true'
    )

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

    parsed_args, unparsed_args = parser.parse_known_args()

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

```

```
        Options:
            -h, --help       Display this help text and exit
            -V, --version    Display the version information and exit

        Any other options will get passed to the QApplication constructor.
        If you don't know what that means, then don't provide any arguments and just the run the program.'''[1:]))
        return

    if parsed_args.version:
        print(dedent(f'''
        lintrans (version {__version__})
        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>'''[1:]))
        return

    for arg in unparsed_args:
        print(f'Passing "{arg}" to QApplication. See --help for recognised args')

    gui.main(args[:1] + unparsed_args)


if __name__ == '__main__':
    main(sys.argv)
```

A.3 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

import re
import sys
import webbrowser
from copy import deepcopy
from typing import List, Tuple, Type

import numpy as np
from numpy import linalg
from numpy.linalg import LinAlgError
from PyQt5 import QtWidgets
from PyQt5.QtCore import pyqtSlot, QCoreApplication, QThread
from PyQt5.QtGui import QCloseEvent, QKeySequence
from PyQt5.QtWidgets import (QApplication, QHBoxLayout, QMainWindow, QMessageBox,
                             QShortcut, QSizePolicy, QSpacerItem, QStyleFactory, QVBoxLayout)

import lintrans
from lintrans.matrices import MatrixWrapper
from lintrans.matrices.parse import validate_matrix_expression
from lintrans.matrices.utility import polar_coords, rotate_coord
from lintrans.typing_ import MatrixType, VectorType
from .dialogs import (AboutDialog, DefineAsAnExpressionDialog, DefineDialog,
                      DefineNumericallyDialog, DefineVisuallyDialog, InfoPanelDialog)
from .dialogs.settings import DisplaySettingsDialog
from .plots import VisualizeTransformationWidget
from .settings import DisplaySettings
from .validate import MatrixExpressionValidator


class LintransMainWindow(QMainWindow):
    """This class provides a main window for the GUI using the Qt framework.
```

```python

    This class should not be used directly, instead call :func:`lintrans.gui.main_window.main` to create the GUI.
    """

    def __init__(self):
        """Create the main window object, and create and arrange every widget in it.

        This doesn't show the window, it just constructs it.
        Use :func:`lintrans.gui.main_window.main` to show the GUI.
        """
        super().__init__()

        self.matrix_wrapper = MatrixWrapper()

        self.setWindowTitle('lintrans')
        self.setMinimumSize(1000, 750)

        self.animating: bool = False
        self.animating_sequence: bool = False

        # === Create menubar

        self.menubar = QtWidgets.QMenuBar(self)

        self.menu_file = QtWidgets.QMenu(self.menubar)
        self.menu_file.setTitle('&File')

        self.menu_help = QtWidgets.QMenu(self.menubar)
        self.menu_help.setTitle('&Help')

        self.action_new = QtWidgets.QAction(self)
        self.action_new.setText('&New')
        self.action_new.setShortcut('Ctrl+N')
        self.action_new.triggered.connect(lambda: print('new'))

        self.action_open = QtWidgets.QAction(self)
        self.action_open.setText('&Open')
        self.action_open.setShortcut('Ctrl+O')
        self.action_open.triggered.connect(lambda: print('open'))

        self.action_save = QtWidgets.QAction(self)
        self.action_save.setText('&Save')
        self.action_save.setShortcut('Ctrl+S')
        self.action_save.triggered.connect(lambda: print('save'))

        self.action_save_as = QtWidgets.QAction(self)
        self.action_save_as.setText('Save as...')
        self.action_save_as.triggered.connect(lambda: print('save as'))

        self.action_tutorial = QtWidgets.QAction(self)
        self.action_tutorial.setText('&Tutorial')
        self.action_tutorial.setShortcut('F1')
        self.action_tutorial.triggered.connect(lambda: print('tutorial'))

        self.action_docs = QtWidgets.QAction(self)
        self.action_docs.setText('&Docs')

        # If this is an old release, use the docs for this release. Else, use the latest docs
        # We use the latest because most use cases for non-stable releases will be in development and testing
        docs_link = 'https://lintrans.readthedocs.io/en/'

        if re.match(r'^\d+\.\d+\.\d+$', lintrans.__version__):
            docs_link += 'v' + lintrans.__version__
        else:
            docs_link += 'latest'

        self.action_docs.triggered.connect(
            lambda: webbrowser.open_new_tab(docs_link)
        )

        self.action_about = QtWidgets.QAction(self)
        self.action_about.setText('&About')
        self.action_about.triggered.connect(lambda: AboutDialog(self).open())
```

```python

        # TODO: Implement these actions and enable them
        self.action_new.setEnabled(False)
        self.action_open.setEnabled(False)
        self.action_save.setEnabled(False)
        self.action_save_as.setEnabled(False)
        self.action_tutorial.setEnabled(False)

        self.menu_file.addAction(self.action_new)
        self.menu_file.addAction(self.action_open)
        self.menu_file.addSeparator()
        self.menu_file.addAction(self.action_save)
        self.menu_file.addAction(self.action_save_as)

        self.menu_help.addAction(self.action_tutorial)
        self.menu_help.addAction(self.action_docs)
        self.menu_help.addSeparator()
        self.menu_help.addAction(self.action_about)

        self.menubar.addAction(self.menu_file.menuAction())
        self.menubar.addAction(self.menu_help.menuAction())

        self.setMenuBar(self.menubar)

        # === Create widgets

        # Left layout: the plot and input box

        self.plot = VisualizeTransformationWidget(self, display_settings=DisplaySettings())

        self.lineedit_expression_box = QtWidgets.QLineEdit(self)
        self.lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
        self.lineedit_expression_box.setValidator(MatrixExpressionValidator(self))
        self.lineedit_expression_box.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')
        # self.button_create_polygon.clicked.connect(self.create_polygon)
        self.button_create_polygon.setToolTip('Define a new polygon to view the transformation of')

        # TODO: Implement this and enable button
        self.button_create_polygon.setEnabled(False)

        self.button_change_display_settings = QtWidgets.QPushButton(self)
        self.button_change_display_settings.setText('Change\ndisplay settings')
        self.button_change_display_settings.clicked.connect(self.dialog_change_display_settings)
        self.button_change_display_settings.setToolTip(
            "Change which things are rendered and how they're rendered<br><b>(Ctrl + D)</b>"
        )
        QShortcut(QKeySequence('Ctrl+D'), self).activated.connect(self.button_change_display_settings.click)

        self.button_reset_zoom = QtWidgets.QPushButton(self)
        self.button_reset_zoom.setText('Reset zoom')
        self.button_reset_zoom.clicked.connect(self.reset_zoom)
        self.button_reset_zoom.setToolTip('Reset the zoom level back to normal<br><b>(Ctrl + Shift + R)</b>')
        QShortcut(QKeySequence('Ctrl+Shift+R'), self).activated.connect(self.button_reset_zoom.click)

        # Define new matrix buttons and their groupbox

        self.button_define_visually = QtWidgets.QPushButton(self)
        self.button_define_visually.setText('Visually')
        self.button_define_visually.setToolTip('Drag the basis vectors<br><b>(Alt + 1)</b>')
        self.button_define_visually.clicked.connect(lambda: self.dialog_define_matrix(DefineVisuallyDialog))
        QShortcut(QKeySequence('Alt+1'), self).activated.connect(self.button_define_visually.click)

        self.button_define_numerically = QtWidgets.QPushButton(self)
        self.button_define_numerically.setText('Numerically')
        self.button_define_numerically.setToolTip('Define a matrix just with numbers<br><b>(Alt + 2)</b>')
        self.button_define_numerically.clicked.connect(lambda: self.dialog_define_matrix(DefineNumericallyDialog))
```

```python
        QShortcut(QKeySequence('Alt+2'), self).activated.connect(self.button_define_numerically.click)

        self.button_define_as_expression = QtWidgets.QPushButton(self)
        self.button_define_as_expression.setText('As an expression')
        self.button_define_as_expression.setToolTip('Define a matrix in terms of other matrices<br><b>(Alt + 3)</b>')
        self.button_define_as_expression.clicked.connect(lambda: self.dialog_define_matrix(DefineAsAnExpressionDialog))
        QShortcut(QKeySequence('Alt+3'), self).activated.connect(self.button_define_as_expression.click)

        self.vlay_define_new_matrix = QVBoxLayout()
        self.vlay_define_new_matrix.setSpacing(20)
        self.vlay_define_new_matrix.addWidget(self.button_define_visually)
        self.vlay_define_new_matrix.addWidget(self.button_define_numerically)
        self.vlay_define_new_matrix.addWidget(self.button_define_as_expression)

        self.groupbox_define_new_matrix = QtWidgets.QGroupBox('Define a new matrix', self)
        self.groupbox_define_new_matrix.setLayout(self.vlay_define_new_matrix)

        # Info panel button

        self.button_info_panel = QtWidgets.QPushButton(self)
        self.button_info_panel.setText('Show defined matrices')
        self.button_info_panel.clicked.connect(
            # We have to use a lambda instead of 'InfoPanelDialog(self.matrix_wrapper, self).open' here
            # because that would create an unnamed instance of InfoPanelDialog when LintransMainWindow is
            # constructed, but we need to create a new instance every time to keep self.matrix_wrapper up to date
            lambda: InfoPanelDialog(self.matrix_wrapper, self).open()
        )
        self.button_info_panel.setToolTip(
            'Open an info panel with all matrices that have been defined in this session<br><b>(Ctrl + M)</b>'
        )
        QShortcut(QKeySequence('Ctrl+M'), self).activated.connect(self.button_info_panel.click)

        # Render buttons

        self.button_reset = QtWidgets.QPushButton(self)
        self.button_reset.setText('Reset')
        self.button_reset.clicked.connect(self.reset_transformation)
        self.button_reset.setToolTip('Reset the visualized transformation back to the identity<br><b>(Ctrl + R)</b>')
        QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(self.button_reset.click)

        self.button_render = QtWidgets.QPushButton(self)
        self.button_render.setText('Render')
        self.button_render.setEnabled(False)
        self.button_render.clicked.connect(self.render_expression)
        self.button_render.setToolTip('Render the expression<br><b>(Ctrl + Enter)</b>')
        QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self.button_render.click)

        self.button_animate = QtWidgets.QPushButton(self)
        self.button_animate.setText('Animate')
        self.button_animate.setEnabled(False)
        self.button_animate.clicked.connect(self.animate_expression)
        self.button_animate.setToolTip('Animate the expression<br><b>(Ctrl + Shift + Enter)</b>')
        QShortcut(QKeySequence('Ctrl+Shift+Return'), self).activated.connect(self.button_animate.click)

        # === Arrange widgets

        self.vlay_left = QVBoxLayout()
        self.vlay_left.addWidget(self.plot)
        self.vlay_left.addWidget(self.lineedit_expression_box)

        self.vlay_misc_buttons = QVBoxLayout()
        self.vlay_misc_buttons.setSpacing(20)
        self.vlay_misc_buttons.addWidget(self.button_create_polygon)
        self.vlay_misc_buttons.addWidget(self.button_change_display_settings)
        self.vlay_misc_buttons.addWidget(self.button_reset_zoom)

        self.vlay_info_buttons = QVBoxLayout()
        self.vlay_info_buttons.setSpacing(20)
        self.vlay_info_buttons.addWidget(self.button_info_panel)

```

```python
        self.vlay_render = QVBoxLayout()
        self.vlay_render.setSpacing(20)
        self.vlay_render.addWidget(self.button_reset)
        self.vlay_render.addWidget(self.button_animate)
        self.vlay_render.addWidget(self.button_render)

        self.vlay_right = QVBoxLayout()
        self.vlay_right.setSpacing(50)
        self.vlay_right.addLayout(self.vlay_misc_buttons)
        self.vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
        self.vlay_right.addWidget(self.groupbox_define_new_matrix)
        self.vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
        self.vlay_right.addLayout(self.vlay_info_buttons)
        self.vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
        self.vlay_right.addLayout(self.vlay_render)

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

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

        self.setCentralWidget(self.central_widget)

    def closeEvent(self, event: QCloseEvent) -> None:
        """Handle a :class:`QCloseEvent` by cancelling animation first."""
        self.animating = False
        event.accept()

    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
        self.plot.update()

    @pyqtSlot()
    def reset_transformation(self) -> None:
        """Reset the visualized transformation back to the identity."""
        self.plot.visualize_matrix_transformation(self.matrix_wrapper['I'])
        self.animating = False
        self.animating_sequence = False
        self.plot.update()

```

```python
    @pyqtSlot()
    def render_expression(self) -> None:
        """Render the transformation given by the expression in the input box."""
        try:
            matrix = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())

        except LinAlgError:
            self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
            return

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

        self.plot.visualize_matrix_transformation(matrix)
        self.plot.update()

    @pyqtSlot()
    def animate_expression(self) -> None:
        """Animate from the current matrix to the matrix in the expression box."""
        self.button_render.setEnabled(False)
        self.button_animate.setEnabled(False)

        matrix_start: MatrixType = np.array([
            [self.plot.point_i[0], self.plot.point_j[0]],
            [self.plot.point_i[1], self.plot.point_j[1]]
        ])

        text = self.lineedit_expression_box.text()

        # If there's commas in the expression, then we want to animate each part at a time
        if ',' in text:
            current_matrix = matrix_start
            self.animating_sequence = True

            # For each expression in the list, right multiply it by the current matrix,
            # and animate from the current matrix to that new matrix
            for expr in text.split(',')[::-1]:
                try:
                    new_matrix = self.matrix_wrapper.evaluate_expression(expr)

                    if self.plot.display_settings.applicative_animation:
                        new_matrix = new_matrix @ current_matrix
                except LinAlgError:
                    self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
                    return

                if not self.animating_sequence:
                    break

                self.animate_between_matrices(current_matrix, new_matrix)
                current_matrix = new_matrix

                # Here we just redraw and allow for other events to be handled while we pause
                self.plot.update()
                QApplication.processEvents()
                QThread.msleep(self.plot.display_settings.animation_pause_length)

            self.animating_sequence = False

        # If there's no commas, then just animate directly from the start to the target
        else:
            # Get the target matrix and it's determinant
            try:
                matrix_target = self.matrix_wrapper.evaluate_expression(text)

            except LinAlgError:
                self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
                return

            # The concept of applicative animation is explained in /gui/settings.py
            if self.plot.display_settings.applicative_animation:
                matrix_target = matrix_target @ matrix_start
```

```python

            # If we want a transitional animation and we're animating the same matrix, then restart the animation
            # We use this check rather than equality because of small floating point errors
            elif (abs(matrix_start - matrix_target) < 1e-12).all():
                matrix_start = self.matrix_wrapper['I']

                # We pause here for 200 ms to make the animation look a bit nicer
                self.plot.visualize_matrix_transformation(matrix_start)
                self.plot.update()
                QApplication.processEvents()
                QThread.msleep(200)

            self.animate_between_matrices(matrix_start, matrix_target)

        self.update_render_buttons()

    def _get_animation_frame(self, start: MatrixType, target: MatrixType, proportion: float) -> MatrixType:
        """Get the matrix to render for this frame of the animation.

        This method will smoothen the determinant if that setting in enabled and if the determinant is positive.
        It also animates rotation-like matrices using a logarithmic spiral to rotate around and scale continuously.
        Essentially, it just makes things look good when animating.

        :param MatrixType start: The starting matrix
        :param MatrixType start: The target matrix
        :param float proportion: How far we are through the loop
        """
        det_target = linalg.det(target)
        det_start = linalg.det(start)

        # This is the matrix that we're applying to get from start to target
        # We want to check if it's rotation-like
        if linalg.det(start) == 0:
            matrix_application = None
        else:
            matrix_application = target @ linalg.inv(start)

        # For a matrix to represent a rotation, it must have a positive determinant,
        # its vectors must be perpendicular, and its vectors must be the same length
        # The checks for 'abs(value) < 1e-10' are to account for floating point error
        if matrix_application is not None \
                and self.plot.display_settings.smoothen_determinant \
                and linalg.det(matrix_application) > 0 \
                and abs(np.dot(matrix_application.T[0], matrix_application.T[1])) < 1e-10 \
                and abs(np.hypot(*matrix_application.T[0]) - np.hypot(*matrix_application.T[1])) < 1e-10:
            rotation_vector: VectorType = matrix_application.T[0]  # Take the i column
            radius, angle = polar_coords(*rotation_vector)

            # We want the angle to be in [-pi, pi), so we have to subtract 2pi from it if it's too big
            if angle > np.pi:
                angle -= 2 * np.pi

            i: VectorType = start.T[0]
            j: VectorType = start.T[1]

            # Scale the coords with a list comprehension
            # It's a bit janky, but rotate_coords() will always return a 2-tuple,
            # so new_i and new_j will always be lists of length 2
            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)

```

```python
        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
                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(self.plot.display_settings.animation_time // steps)

        self.animating = False

    @pyqtSlot(DefineDialog)
    def dialog_define_matrix(self, dialog_class: Type[DefineDialog]) -> None:
        """Open a generic definition dialog to define a new matrix.

        The class for the desired dialog is passed as an argument. We create an
        instance of this class and the dialog is opened asynchronously and modally
        (meaning it blocks interaction with the main window) with the proper method
        connected to the :meth:`QDialog.accepted` signal.
```

```python

        .. note:: ``dialog_class`` must subclass :class:`lintrans.gui.dialogs.define_new_matrix.DefineDialog`.

        :param dialog_class: The dialog class to instantiate
        :type dialog_class: Type[lintrans.gui.dialogs.define_new_matrix.DefineDialog]
        """
        # We create a dialog with a deepcopy of the current matrix_wrapper
        # This avoids the dialog mutating this one
        dialog: DefineDialog

        if dialog_class == DefineVisuallyDialog:
            dialog = DefineVisuallyDialog(
                self,
                matrix_wrapper=deepcopy(self.matrix_wrapper),
                display_settings=self.plot.display_settings
            )
        else:
            dialog = dialog_class(self, matrix_wrapper=deepcopy(self.matrix_wrapper))

        # .open() is asynchronous and doesn't spawn a new event loop, but the dialog is still modal (blocking)
        dialog.open()

        # So we have to use the accepted signal to call a method when the user accepts the dialog
        dialog.accepted.connect(self.assign_matrix_wrapper)

    @pyqtSlot()
    def assign_matrix_wrapper(self) -> None:
        """Assign a new value to ``self.matrix_wrapper`` and give the expression box focus."""
        self.matrix_wrapper = self.sender().matrix_wrapper
        self.lineedit_expression_box.setFocus()
        self.update_render_buttons()

    @pyqtSlot()
    def dialog_change_display_settings(self) -> None:
        """Open the dialog to change the display settings."""
        dialog = DisplaySettingsDialog(self, display_settings=self.plot.display_settings)
        dialog.open()
        dialog.accepted.connect(lambda: self.assign_display_settings(dialog.display_settings))

    @pyqtSlot(DisplaySettings)
    def assign_display_settings(self, display_settings: DisplaySettings) -> None:
        """Assign a new value to ``self.plot.display_settings`` and give the expression box focus."""
        self.plot.display_settings = display_settings
        self.plot.update()
        self.lineedit_expression_box.setFocus()
        self.update_render_buttons()

    def show_error_message(self, title: str, text: str, info: str | None = None) -> None:
        """Show an error message in a dialog box.

        :param str title: The window title of the dialog box
        :param str text: The simple error message
        :param info: The more informative error message
        :type info: Optional[str]
        """
        dialog = QMessageBox(self)
        dialog.setIcon(QMessageBox.Critical)
        dialog.setWindowTitle(title)
        dialog.setText(text)

        if info is not None:
            dialog.setInformativeText(info)

        dialog.open()

        # This is `finished` rather than `accepted` because we want to update the buttons no matter what
        dialog.finished.connect(self.update_render_buttons)

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

        Convert the elements of the matrix to canvas coords and make sure they fit within Qt's 32-bit integer limit.

```

```
        :param MatrixType matrix: The matrix to check
        :returns bool: Whether the matrix is too big to fit on the canvas
        """
        coords: List[Tuple[int, int]] = [self.plot.canvas_coords(*vector) for vector in matrix.T]

        for x, y in coords:
            if not (-2147483648 <= x <= 2147483647 and -2147483648 <= y <= 2147483647):
                return True

        return False


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


def main(args: List[str]) -> None:
    """Run the GUI by creating and showing an instance of :class:`LintransMainWindow`.

    :param List[str] args: The args to pass to :class:`QApplication`
    """
    app = QApplication(args)
    app.setApplicationName('lintrans')
    app.setApplicationVersion(lintrans.__version__)

    qapp().setStyle(QStyleFactory.create('fusion'))

    window = LintransMainWindow()
    window.show()

    sys.exit(app.exec_())
```

A.4 gui/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>

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

from __future__ import annotations

from dataclasses import dataclass


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

    # === Basic stuff

    draw_background_grid: bool = True
    """This controls whether we want to draw the background grid.

    The background axes will always be drawn. This makes it easy to identify the center of the space.
    """

    draw_transformed_grid: bool = True
    """This controls whether we want to draw the transformed grid. Vectors are handled separately."""
```

```python

    draw_basis_vectors: bool = True
    """This controls whether we want to draw the transformed basis vectors."""

    # === Animations

    smoothen_determinant: bool = True
    """This controls whether we want the determinant to change smoothly during the animation.

    .. note::
       Even if this is True, it will be ignored if we're animating from a positive det matrix to
       a negative det matrix, or vice versa, because if we try to smoothly animate that determinant,
       things blow up and the app often crashes.
    """

    applicative_animation: bool = True
    """There are two types of simple animation, transitional and applicative.

    Let ``C`` be the matrix representing the currently displayed transformation, and let ``T`` be the target matrix.
    Transitional animation means that we animate directly from ``C`` from ``T``,
    and applicative animation means that we animate from ``C`` to ``TC``, so we apply ``T`` to ``C``.
    """

    animation_time: int = 1200
    """This is the number of milliseconds that an animation takes."""

    animation_pause_length: int = 400
    """This is the number of milliseconds that we wait between animations when using comma syntax."""

    # === Matrix info

    draw_determinant_parallelogram: bool = False
    """This controls whether or not we should shade the parallelogram representing the determinant of the matrix."""

    show_determinant_value: bool = True
    """This controls whether we should write the text value of the determinant inside the parallelogram.

    The text only gets draw if :attr:`draw_determinant_parallelogram` is also True.
    """

    draw_eigenvectors: bool = False
    """This controls whether we should draw the eigenvectors of the transformation."""

    draw_eigenlines: bool = False
    """This controls whether we should draw the eigenlines of the transformation."""
```

A.5 gui/__init__.py

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

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

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

A.6 gui/validate.py

```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>
```

```python

"""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."""
        clean_text = re.sub(parse.NAIVE_CHARACTER_CLASS[:-1] + ',]', '', text)

        if clean_text == '':
            if parse.validate_matrix_expression(clean_text):
                return QValidator.Acceptable, text, pos
            else:
                return QValidator.Intermediate, text, pos

        return QValidator.Invalid, text, pos
```

A.7 gui/dialogs/misc.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>

"""This module provides miscellaneous dialog classes like :class:`AboutDialog`."""

from __future__ import annotations

import platform
from typing import Union

from PyQt5.QtCore import PYQT_VERSION_STR, QT_VERSION_STR, Qt
from PyQt5.QtWidgets import QDialog, QGridLayout, QLabel, QVBoxLayout, QWidget

import lintrans
from lintrans.matrices.utility import round_float
from lintrans.matrices import MatrixWrapper
from lintrans.typing_ import is_matrix_type, MatrixType


class FixedSizeDialog(QDialog):
    """A simple superclass to create modal dialog boxes with fixed size.

    We override the :meth:`open` method to set the fixed size as soon as the dialog is opened modally.
    """

    def open(self) -> None:
        """Override :meth:`QDialog.open` to set the dialog to a fixed size."""
        super().open()
        self.setFixedSize(self.size())


class AboutDialog(FixedSizeDialog):
    """A simple dialog class to display information about the app to the user.

    It only has an :meth:`__init__` method because it only has label widgets, so no other methods are necessary
↪   here.
    """
```

```python

    def __init__(self, *args, **kwargs):
        """Create an :class:`AboutDialog` object with all the label widgets."""
        super().__init__(*args, **kwargs)

        self.setWindowTitle('About lintrans')

        # === Create the widgets

        label_title = QLabel(self)
        label_title.setText(f'lintrans (version {lintrans.__version__})')
        label_title.setAlignment(Qt.AlignCenter)

        font_title = label_title.font()
        font_title.setPointSize(font_title.pointSize() * 2)
        label_title.setFont(font_title)

        label_version_info = QLabel(self)
        label_version_info.setText(
            f'With Python version {platform.python_version()}\n'
            f'Qt version {QT_VERSION_STR} and PyQt5 version {PYQT_VERSION_STR}\n'
            f'Running on {platform.platform()}'
        )
        label_version_info.setAlignment(Qt.AlignCenter)

        label_info = QLabel(self)
        label_info.setText(
            'lintrans is a program designed to help visualise<br>'
            '2D linear transformations represented with matrices.<br><br>'
            "It's designed for teachers and students and any feedback<br>"
            'is greatly appreciated at <a href="https://github.com/DoctorDalek1963/lintrans" '
            'style="color: black;">my GitHub page</a><br>or via email '
            '(<a href="mailto:dyson.dyson@icloud.com" style="color: black;">dyson.dyson@icloud.com</a>).'
        )
        label_info.setAlignment(Qt.AlignCenter)
        label_info.setTextFormat(Qt.RichText)
        label_info.setOpenExternalLinks(True)

        label_copyright = QLabel(self)
        label_copyright.setText(
            'This program is free software.<br>Copyright 2021-2022 D. Dyson (DoctorDalek1963).<br>'
            'This program is licensed under GPLv3, which can be found '
            '<a href="https://www.gnu.org/licenses/gpl-3.0.html" style="color: black;">here</a>.'
        )
        label_copyright.setAlignment(Qt.AlignCenter)
        label_copyright.setTextFormat(Qt.RichText)
        label_copyright.setOpenExternalLinks(True)

        # === Arrange the widgets

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

        vlay = QVBoxLayout()
        vlay.setSpacing(20)
        vlay.addWidget(label_title)
        vlay.addWidget(label_version_info)
        vlay.addWidget(label_info)
        vlay.addWidget(label_copyright)

        self.setLayout(vlay)


class InfoPanelDialog(FixedSizeDialog):
    """A simple dialog class to display an info panel that shows all currently defined matrices."""

    def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
        """Create the dialog box with all the widgets needed to show the information."""
        super().__init__(*args, **kwargs)
        self.wrapper = matrix_wrapper

        self.setWindowTitle('Defined matrices')

        grid_layout = QGridLayout()
```

```
            grid_layout.setSpacing(20)

            bold_font = self.font()
            bold_font.setBold(True)

            name_value_pair: tuple[str, Union[MatrixType, str]]

            # Each defined matrix will get a widget group. Each group will be a label for the name,
            # a label for '=', and a container widget to either show the matrix numerically, or to
            # show the expression that it's defined as
            for i, name_value_pair in enumerate(self.wrapper.get_defined_matrices()):
                name, value = name_value_pair

                # Create all the widgets first
                label_name = QLabel(self)
                label_name.setText(name)
                label_name.setFont(bold_font)

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

                widget_matrix = self._get_matrix_widget(value)

                # We want columns of at most 6 widget groups
                # This column variable manages which column of defined matrices we're on
                # It's multiplied by 3 because all the widgets are in a single grid layout
                # I could factor out each triplet of widgets for a defined matrix into a container widget,
                # but I prefer to keep the widget count lower to reduce any possible lag
                column = 3 * (i // 6)

                grid_layout.addWidget(
                    label_name,
                    i - 2 * column,
                    column,
                    Qt.AlignCenter
                )
                grid_layout.addWidget(
                    label_equals,
                    i - 2 * column,
                    column + 1,
                    Qt.AlignCenter
                )
                grid_layout.addWidget(
                    widget_matrix,
                    i - 2 * column,
                    column + 2,
                    Qt.AlignCenter
                )

            self.setContentsMargins(10, 10, 10, 10)
            self.setLayout(grid_layout)

        def _get_matrix_widget(self, matrix: Union[MatrixType, str]) -> QWidget:
            """Return a :class:`QWidget` containing the value of the matrix.

            If the matrix is defined as an expression, it will be a simple :class:`QLabel`.
            If the matrix is defined as a matrix, it will be a :class:`QWidget` container
            with multiple :class:`QLabel` objects in it.
            """
            if isinstance(matrix, str):
                label = QLabel(self)
                label.setText(matrix)
                return label

            elif is_matrix_type(matrix):
                # tl = top left, br = bottom right, etc.
                label_tl = QLabel(self)
                label_tl.setText(round_float(matrix[0][0]))

                label_tr = QLabel(self)
                label_tr.setText(round_float(matrix[0][1]))

                label_bl = QLabel(self)
```

```python
            label_bl.setText(round_float(matrix[1][0]))

            label_br = QLabel(self)
            label_br.setText(round_float(matrix[1][1]))

            # The parens need to be bigger than the numbers, but increasing the font size also
            # makes the font thicker, so we have to reduce the font weight by the same factor
            font_parens = self.font()
            font_parens.setPointSize(int(font_parens.pointSize() * 2.5))
            font_parens.setWeight(int(font_parens.weight() / 2.5))

            label_paren_left = QLabel(self)
            label_paren_left.setText('(')
            label_paren_left.setFont(font_parens)

            label_paren_right = QLabel(self)
            label_paren_right.setText(')')
            label_paren_right.setFont(font_parens)

            container = QWidget(self)
            grid_layout = QGridLayout()

            grid_layout.addWidget(label_paren_left, 0, 0, -1, 1)
            grid_layout.addWidget(label_tl, 0, 1)
            grid_layout.addWidget(label_tr, 0, 2)
            grid_layout.addWidget(label_bl, 1, 1)
            grid_layout.addWidget(label_br, 1, 2)
            grid_layout.addWidget(label_paren_right, 0, 3, -1, 1)

            container.setLayout(grid_layout)

            return container

        raise ValueError('Matrix was not MatrixType or str')
```

A.8 gui/dialogs/settings.py

```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 dialogs to edit settings within the app."""

from __future__ import annotations

import abc
from typing import Dict

from PyQt5 import QtWidgets
from PyQt5.QtGui import QIntValidator, QKeyEvent, QKeySequence
from PyQt5.QtWidgets import QCheckBox, QGroupBox, QHBoxLayout, QShortcut, QSizePolicy, QSpacerItem, QVBoxLayout

from lintrans.gui.dialogs.misc import FixedSizeDialog
from lintrans.gui.settings import DisplaySettings


class SettingsDialog(FixedSizeDialog):
    """An abstract superclass for other simple dialogs."""

    def __init__(self, *args, **kwargs):
        """Create the widgets and layout of the dialog, passing ``*args`` and ``**kwargs`` to super."""
        super().__init__(*args, **kwargs)

        # === Create the widgets

        self.button_confirm = QtWidgets.QPushButton(self)
        self.button_confirm.setText('Confirm')
        self.button_confirm.clicked.connect(self.confirm_settings)
```

```
        self.button_confirm.setToolTip('Confirm these new settings<br><b>(Ctrl + Enter)</b>')
        QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self.button_confirm.click)

        self.button_cancel = QtWidgets.QPushButton(self)
        self.button_cancel.setText('Cancel')
        self.button_cancel.clicked.connect(self.reject)
        self.button_cancel.setToolTip('Revert these settings<br><b>(Escape)</b>')

        # === 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(self.button_cancel)
        self.hlay_buttons.addWidget(self.button_confirm)

        self.vlay_options = QVBoxLayout()
        self.vlay_options.setSpacing(20)

        self.vlay_all = QVBoxLayout()
        self.vlay_all.setSpacing(20)
        self.vlay_all.addLayout(self.vlay_options)
        self.vlay_all.addLayout(self.hlay_buttons)

        self.setLayout(self.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."""


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, **kwargs)

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

        self.dict_checkboxes: Dict[str, QCheckBox] = dict()

        # === 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(
```

```python
            'Draw the transformed basis vectors'
        )
        self.dict_checkboxes['v'] = self.checkbox_draw_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

        self.label_animation_time = QtWidgets.QLabel(self)
        self.label_animation_time.setText('Total animation length (ms)')
        self.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.label_animation_pause_length = QtWidgets.QLabel(self)
        self.label_animation_pause_length.setText('Animation pause length (ms)')
        self.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')
        self.dict_checkboxes['l'] = self.checkbox_draw_eigenlines

        # === Arrange the widgets in QGroupBoxes

        # Basic stuff

        self.vlay_groupbox_basic_stuff = QVBoxLayout()
        self.vlay_groupbox_basic_stuff.setSpacing(20)
        self.vlay_groupbox_basic_stuff.addWidget(self.checkbox_draw_background_grid)
```

```python
        self.vlay_groupbox_basic_stuff.addWidget(self.checkbox_draw_transformed_grid)
        self.vlay_groupbox_basic_stuff.addWidget(self.checkbox_draw_basis_vectors)

        self.groupbox_basic_stuff = QGroupBox('Basic stuff', self)
        self.groupbox_basic_stuff.setLayout(self.vlay_groupbox_basic_stuff)

        # Animations

        self.hlay_animation_time = QHBoxLayout()
        self.hlay_animation_time.addWidget(self.label_animation_time)
        self.hlay_animation_time.addWidget(self.lineedit_animation_time)

        self.hlay_animation_pause_length = QHBoxLayout()
        self.hlay_animation_pause_length.addWidget(self.label_animation_pause_length)
        self.hlay_animation_pause_length.addWidget(self.lineedit_animation_pause_length)

        self.vlay_groupbox_animations = QVBoxLayout()
        self.vlay_groupbox_animations.setSpacing(20)
        self.vlay_groupbox_animations.addWidget(self.checkbox_smoothen_determinant)
        self.vlay_groupbox_animations.addWidget(self.checkbox_applicative_animation)
        self.vlay_groupbox_animations.addLayout(self.hlay_animation_time)
        self.vlay_groupbox_animations.addLayout(self.hlay_animation_pause_length)

        self.groupbox_animations = QGroupBox('Animations', self)
        self.groupbox_animations.setLayout(self.vlay_groupbox_animations)

        # Matrix info

        self.vlay_groupbox_matrix_info = QVBoxLayout()
        self.vlay_groupbox_matrix_info.setSpacing(20)
        self.vlay_groupbox_matrix_info.addWidget(self.checkbox_draw_determinant_parallelogram)
        self.vlay_groupbox_matrix_info.addWidget(self.checkbox_show_determinant_value)
        self.vlay_groupbox_matrix_info.addWidget(self.checkbox_draw_eigenvectors)
        self.vlay_groupbox_matrix_info.addWidget(self.checkbox_draw_eigenlines)

        self.groupbox_matrix_info = QGroupBox('Matrix info', self)
        self.groupbox_matrix_info.setLayout(self.vlay_groupbox_matrix_info)

        # Now arrange the groupboxes
        self.vlay_options.addWidget(self.groupbox_basic_stuff)
        self.vlay_options.addWidget(self.groupbox_animations)
        self.vlay_options.addWidget(self.groupbox_matrix_info)

        # Finally, we load the current settings and update the GUI
        self.load_settings()
        self.update_gui()

    def load_settings(self) -> None:
        """Load the current display settings into the widgets."""
        # Basic stuff
        self.checkbox_draw_background_grid.setChecked(self.display_settings.draw_background_grid)
        self.checkbox_draw_transformed_grid.setChecked(self.display_settings.draw_transformed_grid)
        self.checkbox_draw_basis_vectors.setChecked(self.display_settings.draw_basis_vectors)

        # Animations
        self.checkbox_smoothen_determinant.setChecked(self.display_settings.smoothen_determinant)
        self.checkbox_applicative_animation.setChecked(self.display_settings.applicative_animation)
        self.lineedit_animation_time.setText(str(self.display_settings.animation_time))
        self.lineedit_animation_pause_length.setText(str(self.display_settings.animation_pause_length))

        # Matrix info
        self.checkbox_draw_determinant_parallelogram.setChecked(
            self.display_settings.draw_determinant_parallelogram)
        self.checkbox_show_determinant_value.setChecked(self.display_settings.show_determinant_value)
        self.checkbox_draw_eigenvectors.setChecked(self.display_settings.draw_eigenvectors)
        self.checkbox_draw_eigenlines.setChecked(self.display_settings.draw_eigenlines)

    def confirm_settings(self) -> None:
        """Build a :class:`lintrans.gui.settings.DisplaySettings` object and assign it."""
        # Basic stuff
        self.display_settings.draw_background_grid = self.checkbox_draw_background_grid.isChecked()
        self.display_settings.draw_transformed_grid = self.checkbox_draw_transformed_grid.isChecked()
        self.display_settings.draw_basis_vectors = self.checkbox_draw_basis_vectors.isChecked()
```

```python

        # Animations
        self.display_settings.smoothen_determinant = self.checkbox_smoothen_determinant.isChecked()
        self.display_settings.applicative_animation = self.checkbox_applicative_animation.isChecked()
        self.display_settings.animation_time = int(self.lineedit_animation_time.text())
        self.display_settings.animation_pause_length = int(self.lineedit_animation_pause_length.text())

        # Matrix info
        self.display_settings.draw_determinant_parallelogram = self.checkbox_draw_determinant_parallelogram.isChecked()
        self.display_settings.show_determinant_value = self.checkbox_show_determinant_value.isChecked()
        self.display_settings.draw_eigenvectors = self.checkbox_draw_eigenvectors.isChecked()
        self.display_settings.draw_eigenlines = self.checkbox_draw_eigenlines.isChecked()

        self.accept()

    def update_gui(self) -> None:
        """Update the GUI according to other widgets in the GUI.

        For example, this method updates which checkboxes are enabled based on the values of other checkboxes.
        """
        self.checkbox_show_determinant_value.setEnabled(self.checkbox_draw_determinant_parallelogram.isChecked())

    def keyPressEvent(self, event: QKeyEvent) -> None:
        """Handle a :class:`QKeyEvent` by manually activating toggling checkboxes.

        Qt handles these shortcuts automatically and allows the user to do ``Alt + Key``
        to activate a simple shortcut defined with ``&``. However, I like to be able to
        just hit ``Key`` and have the shortcut activate.
        """
        letter = event.text().lower()
        key = event.key()

        if letter in self.dict_checkboxes:
            self.dict_checkboxes[letter].animateClick()

        # Return or keypad enter
        elif key == 0x01000004 or key == 0x01000005:
            self.button_confirm.click()

        # Escape
        elif key == 0x01000000:
            self.button_cancel.click()

        else:
            event.ignore()
```

A.9 gui/dialogs/__init__.py

```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 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 DefineAsAnExpressionDialog, DefineDialog, DefineNumericallyDialog, DefineVisuallyDialog
from .misc import AboutDialog, InfoPanelDialog
from .settings import DisplaySettingsDialog

__all__ = ['AboutDialog', 'DefineAsAnExpressionDialog', 'DefineDialog', 'DefineNumericallyDialog',
           'DefineVisuallyDialog', 'DisplaySettingsDialog', 'InfoPanelDialog']
```

A.10 gui/dialogs/define_new_matrix.py

```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 an abstract :class:`DefineDialog` class and subclasses, allowing definition of new matrices."""

from __future__ import annotations

import abc

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, QShortcut, QSizePolicy, QSpacerItem, QVBoxLayout

from lintrans.gui.dialogs.misc import FixedSizeDialog
from lintrans.gui.plots import DefineVisuallyWidget
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 DefineDialog(FixedSizeDialog):
    """An abstract superclass for definitions dialogs.

    .. 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 :class:`QDialog`,
       and every superclass of a class must have the same metaclass, and :class:`QDialog` is not an abstract class.
    """

    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 = QtWidgets.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)

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

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

        self.combobox_letter = QtWidgets.QComboBox(self)
```

```python

        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(self.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(self.label_equals)

        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(DefineDialog):
    """The dialog class that allows the user to define a matrix visually."""

    def __init__(self, *args, matrix_wrapper: MatrixWrapper, display_settings: DisplaySettings, **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

        self.plot = DefineVisuallyWidget(self, display_settings=display_settings)
```

```python

        # === Arrange the widgets

        self.hlay_definition.addWidget(self.plot)
        self.hlay_definition.setStretchFactor(self.plot, 1)

        self.vlay_all.addLayout(self.hlay_definition)
        self.vlay_all.addLayout(self.hlay_buttons)

        # We load the default matrix A into the plot
        self.load_matrix(0)

        # We also enable the confirm button, because any visually defined matrix is valid
        self.button_confirm.setEnabled(True)

    @pyqtSlot()
    def update_confirm_button(self) -> None:
        """Enable the confirm button.

        .. note::
           The confirm button is always enabled in this dialog and this method is never actually used,
           so it's got an empty body. It's only here because we need to implement the abstract method.
        """

    @pyqtSlot(int)
    def load_matrix(self, index: int) -> None:
        """Show the selected matrix on the plot. If the matrix is None, show the identity."""
        matrix = self.matrix_wrapper[self.selected_letter]

        if matrix is None:
            matrix = self.matrix_wrapper['I']

        self.plot.visualize_matrix_transformation(matrix)
        self.plot.update()

    @pyqtSlot()
    def confirm_matrix(self) -> None:
        """Confirm the matrix that's been defined visually."""
        matrix: MatrixType = array([
            [self.plot.point_i[0], self.plot.point_j[0]],
            [self.plot.point_i[1], self.plot.point_j[1]]
        ])

        self.matrix_wrapper[self.selected_letter] = matrix
        self.accept()


class DefineNumericallyDialog(DefineDialog):
    """The dialog class that allows the user to define a new matrix numerically."""

    def __init__(self, *args, matrix_wrapper: MatrixWrapper, **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)

        # === Create the widgets

        # tl = top left, br = bottom right, etc.
        self.element_tl = QtWidgets.QLineEdit(self)
        self.element_tl.textChanged.connect(self.update_confirm_button)
        self.element_tl.setValidator(QDoubleValidator())

        self.element_tr = QtWidgets.QLineEdit(self)
        self.element_tr.textChanged.connect(self.update_confirm_button)
        self.element_tr.setValidator(QDoubleValidator())

        self.element_bl = QtWidgets.QLineEdit(self)
        self.element_bl.textChanged.connect(self.update_confirm_button)
        self.element_bl.setValidator(QDoubleValidator())

        self.element_br = QtWidgets.QLineEdit(self)
```

```
        self.element_br.textChanged.connect(self.update_confirm_button)
        self.element_br.setValidator(QDoubleValidator())

        self.matrix_elements = (self.element_tl, self.element_tr, self.element_bl, self.element_br)

        # === Arrange the widgets

        self.grid_matrix = QGridLayout()
        self.grid_matrix.setSpacing(20)
        self.grid_matrix.addWidget(self.element_tl, 0, 0)
        self.grid_matrix.addWidget(self.element_tr, 0, 1)
        self.grid_matrix.addWidget(self.element_bl, 1, 0)
        self.grid_matrix.addWidget(self.element_br, 1, 1)

        self.hlay_definition.addLayout(self.grid_matrix)

        self.vlay_all.addLayout(self.hlay_definition)
        self.vlay_all.addLayout(self.hlay_buttons)

        # We load the default matrix A into the boxes
        self.load_matrix(0)

        self.element_tl.setFocus()

    @pyqtSlot()
    def update_confirm_button(self) -> None:
        """Enable the confirm button if there are valid floats in every box."""
        for elem in self.matrix_elements:
            if not is_valid_float(elem.text()):
                # If they're not all numbers, then we can't confirm it
                self.button_confirm.setEnabled(False)
                return

        # If we didn't find anything invalid
        self.button_confirm.setEnabled(True)

    @pyqtSlot(int)
    def load_matrix(self, index: int) -> None:
        """If the selected matrix is defined, load its values into the boxes."""
        matrix = self.matrix_wrapper[self.selected_letter]

        if matrix is None:
            for elem in self.matrix_elements:
                elem.setText('')

        else:
            self.element_tl.setText(round_float(matrix[0][0]))
            self.element_tr.setText(round_float(matrix[0][1]))
            self.element_bl.setText(round_float(matrix[1][0]))
            self.element_br.setText(round_float(matrix[1][1]))

        self.update_confirm_button()

    @pyqtSlot()
    def confirm_matrix(self) -> None:
        """Confirm the matrix in the boxes and assign it to the name in the combo box."""
        matrix: MatrixType = array([
            [float(self.element_tl.text()), float(self.element_tr.text())],
            [float(self.element_bl.text()), float(self.element_br.text())]
        ])

        self.matrix_wrapper[self.selected_letter] = matrix
        self.accept()


class DefineAsAnExpressionDialog(DefineDialog):
    """The dialog class that allows the user to define a matrix as an expression of other matrices."""

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

        :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
        """
```

```python
        super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)

        self.setMinimumWidth(450)

        # === Create the widgets

        self.lineedit_expression_box = QtWidgets.QLineEdit(self)
        self.lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
        self.lineedit_expression_box.textChanged.connect(self.update_confirm_button)
        self.lineedit_expression_box.setValidator(MatrixExpressionValidator())

        # === Arrange the widgets

        self.hlay_definition.addWidget(self.lineedit_expression_box)

        self.vlay_all.addLayout(self.hlay_definition)
        self.vlay_all.addLayout(self.hlay_buttons)

        # Load the matrix if it's defined as an expression
        self.load_matrix(0)

        self.lineedit_expression_box.setFocus()

    @pyqtSlot()
    def update_confirm_button(self) -> None:
        """Enable the confirm button if the matrix expression is valid in the wrapper."""
        text = self.lineedit_expression_box.text()
        valid_expression = self.matrix_wrapper.is_valid_expression(text)

        self.button_confirm.setEnabled(valid_expression and self.selected_letter not in text)

    @pyqtSlot(int)
    def load_matrix(self, index: int) -> None:
        """If the selected matrix is defined an expression, load that expression into the box."""
        if (expr := self.matrix_wrapper.get_expression(self.selected_letter)) is not None:
            self.lineedit_expression_box.setText(expr)
        else:
            self.lineedit_expression_box.setText('')

    @pyqtSlot()
    def confirm_matrix(self) -> None:
        """Evaluate the matrix expression and assign its value to the name in the combo box."""
        self.matrix_wrapper[self.selected_letter] = self.lineedit_expression_box.text()
        self.accept()
```

A.11 gui/plots/__init__.py

```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 package provides widgets for the visualization plot in the main window and the visual definition dialog."""

from .classes import BackgroundPlot, VectorGridPlot
from .widgets import DefineVisuallyWidget, VisualizeTransformationWidget

__all__ = ['BackgroundPlot', 'DefineVisuallyWidget', 'VectorGridPlot', 'VisualizeTransformationWidget']
```

A.12 gui/plots/widgets.py

```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>

```

```python
"""This module provides the actual widgets that can be used to visualize transformations in the GUI."""

from __future__ import annotations

from math import ceil, dist, floor
from typing import List, Tuple

from PyQt5.QtCore import Qt
from PyQt5.QtGui import QMouseEvent, QPainter, QPaintEvent

from lintrans.typing_ import MatrixType
from lintrans.gui.settings import DisplaySettings
from .classes import VectorGridPlot


class VisualizeTransformationWidget(VectorGridPlot):
    """This class is the widget that is used in the main window to visualize transformations.

    It handles all the rendering itself, and the only method that the user needs to
    worry about is :meth:`visualize_matrix_transformation`, which allows you to visualize
    the given matrix transformation.
    """

    def __init__(self, *args, display_settings: DisplaySettings, **kwargs):
        """Create the widget and assign its display settings, passing ``*args`` and ``**kwargs`` to super."""
        super().__init__(*args, **kwargs)

        self.display_settings = display_settings

    def visualize_matrix_transformation(self, matrix: MatrixType) -> None:
        """Transform the grid by the given matrix.

        .. warning:: This method does not call ``update()``. This must be done by the caller.

        .. note::
           This method transforms the background grid, not the basis vectors. This
           means that it cannot be used to compose transformations. Compositions
           should be done by the user.

        :param MatrixType matrix: The matrix to transform by
        """
        self.point_i = (matrix[0][0], matrix[1][0])
        self.point_j = (matrix[0][1], matrix[1][1])

    def paintEvent(self, event: QPaintEvent) -> None:
        """Handle a :class:`QPaintEvent` by drawing the background grid and the transformed grid.

        The transformed grid is defined by the basis vectors i and j, which can
        be controlled with the :meth:`visualize_matrix_transformation` method.
        """
        painter = QPainter()
        painter.begin(self)

        painter.setRenderHint(QPainter.Antialiasing)
        painter.setBrush(Qt.NoBrush)

        self.draw_background(painter, self.display_settings.draw_background_grid)

        if self.display_settings.draw_eigenlines:
            self.draw_eigenlines(painter)

        if self.display_settings.draw_eigenvectors:
            self.draw_eigenvectors(painter)

        if self.display_settings.draw_determinant_parallelogram:
            self.draw_determinant_parallelogram(painter)

            if self.display_settings.show_determinant_value:
                self.draw_determinant_text(painter)

        if self.display_settings.draw_transformed_grid:
            self.draw_transformed_grid(painter)

```

```python
        if self.display_settings.draw_basis_vectors:
            self.draw_basis_vectors(painter)

        painter.end()
        event.accept()


class DefineVisuallyWidget(VisualizeTransformationWidget):
    """This class is the widget that allows the user to visually define a matrix.

    This is just the widget itself. If you want the dialog, use
    :class:`lintrans.gui.dialogs.define_new_matrix.DefineVisuallyDialog`.
    """

    def __init__(self, *args, display_settings: DisplaySettings, **kwargs):
        """Create the widget and enable mouse tracking. ``*args`` and ``**kwargs`` are passed to ``super()``."""
        super().__init__(*args, display_settings=display_settings, **kwargs)

        self.dragged_point: Tuple[float, float] | None = None

        # This is the distance that the cursor needs to be from the point to drag it
        self.epsilon: int = 5

    def mousePressEvent(self, event: QMouseEvent) -> None:
        """Handle a :class:`QMouseEvent` when the user presses a button."""
        mx = event.x()
        my = event.y()
        button = event.button()

        if button != Qt.LeftButton:
            event.ignore()
            return

        for point in (self.point_i, self.point_j):
            px, py = self.canvas_coords(*point)
            if abs(px - mx) <= self.epsilon and abs(py - my) <= self.epsilon:
                self.dragged_point = point[0], point[1]

        event.accept()

    def mouseReleaseEvent(self, event: QMouseEvent) -> None:
        """Handle a :class:`QMouseEvent` when the user releases a button."""
        if event.button() == Qt.LeftButton:
            self.dragged_point = None
            event.accept()
        else:
            event.ignore()

    def mouseMoveEvent(self, event: QMouseEvent) -> None:
        """Handle the mouse moving on the canvas."""
        mx = event.x()
        my = event.y()

        if self.dragged_point is None:
            event.ignore()
            return

        x, y = self.grid_coords(mx, my)

        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 < 0.1:
```

```python
                x, y = coord

        if self.dragged_point == self.point_i:
            self.point_i = x, y

        elif self.dragged_point == self.point_j:
            self.point_j = x, y

        self.dragged_point = x, y

        self.update()

        event.accept()
```

A.13 gui/plots/classes.py

```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 superclasses for plotting transformations."""

from __future__ import annotations

from abc import abstractmethod
from typing import Iterable, List, Tuple

import numpy as np
from nptyping import Float, NDArray
from PyQt5.QtCore import QPoint, QRectF, Qt
from PyQt5.QtGui import QBrush, QColor, QPainter, QPainterPath, QPaintEvent, QPen, QWheelEvent
from PyQt5.QtWidgets import QWidget

from lintrans.typing_ import MatrixType


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

    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 :class:`QWidget`,
       and every superclass of a class must have the same metaclass, and :class:`QWidget` is not an abstract class.
    """

    default_grid_spacing: int = 85
    minimum_grid_spacing: int = 5

    def __init__(self, *args, **kwargs):
        """Create the widget and setup backend stuff for rendering.

        .. note:: ``*args`` and ``**kwargs`` are passed the superclass constructor (:class:`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)

```

```python
        # Set the grid colour to grey and the axes colour to black
        self.colour_background_grid = QColor('#808080')
        self.colour_background_axes = QColor('#000000')

        self.grid_spacing = BackgroundPlot.default_grid_spacing
        self.width_background_grid: float = 0.3

    @property
    def canvas_origin(self) -> Tuple[int, int]:
        """Return the canvas coords of the grid origin.

        The return value is intended to be unpacked and passed to a :meth:`QPainter.drawLine:iiii` call.

        See :meth:`canvas_coords`.

        :returns: The canvas coordinates of the grid origin
        :rtype: Tuple[int, int]
        """
        return self.width() // 2, self.height() // 2

    def canvas_x(self, x: float) -> int:
        """Convert an x coordinate from grid coords to canvas coords."""
        return int(self.canvas_origin[0] + x * self.grid_spacing)

    def canvas_y(self, y: float) -> int:
        """Convert a y coordinate from grid coords to canvas coords."""
        return int(self.canvas_origin[1] - y * self.grid_spacing)

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

        This method is intended to be used like

        .. code::

           painter.drawLine(*self.canvas_coords(x1, y1), *self.canvas_coords(x2, y2))

        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.
        """
```

```python

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

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

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

    def __init__(self, *args, **kwargs):
        """Create the widget with ``point_i`` and ``point_j`` attributes.

        .. note:: ``*args`` and ``**kwargs`` are passed to the superclass constructor (:class:`BackgroundPlot`).
        """
        super().__init__(*args, **kwargs)

        self.point_i: Tuple[float, float] = (1., 0.)
        self.point_j: Tuple[float, float] = (0., 1.)

        self.colour_i = QColor('#0808d8')
        self.colour_j = QColor('#e90000')
        self.colour_eigen = QColor('#13cf00')
        self.colour_text = QColor('#000000')

        self.width_vector_line = 1.8
        self.width_transformed_grid = 0.8

        self.arrowhead_length = 0.15

        self.max_parallel_lines = 150
```

```python

    @property
    def matrix(self) -> MatrixType:
        """Return the assembled matrix of the basis vectors."""
        return np.array([
            [self.point_i[0], self.point_j[0]],
            [self.point_i[1], self.point_j[1]]
        ])

    @property
    def det(self) -> float:
        """Return the determinant of the assembled matrix."""
        return float(np.linalg.det(self.matrix))

    @property
    def eigs(self) -> Iterable[Tuple[float, NDArray[(1, 2), Float]]]:
        """Return the eigenvalues and eigenvectors zipped together to be iterated over.

        :rtype: Iterable[Tuple[float, NDArray[(1, 2), Float]]]
        """
        values, vectors = np.linalg.eig(self.matrix)
        return zip(values, vectors.T)

    @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_parallel_lines(self, painter: QPainter, vector: Tuple[float, float], point: Tuple[float, float]) ->
    ↪ None:
        """Draw a set of evenly spaced grid lines parallel to ``vector`` intersecting ``point``.

        :param QPainter painter: The painter to draw the lines with
        :param vector: The vector to draw the grid lines parallel to
        :type vector: Tuple[float, float]
        :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
```

```python

        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(max(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(max(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.

```

```python
        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]),
                *self.canvas_coords(*points[1])
            )
            return True

    def draw_transformed_grid(self, painter: QPainter) -> None:
        """Draw the transformed version of the grid, given by the basis vectors.

        .. note:: This method draws the grid, but not the basis vectors. Use :meth:`draw_basis_vectors` to draw them.

        :param QPainter painter: The painter to draw the grid lines with
        """
        # 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)

    def draw_arrowhead_away_from_origin(self, painter: QPainter, point: Tuple[float, float]) -> None:
        """Draw an arrowhead at ``point``, pointing away from the origin.

        :param QPainter painter: The painter to draw the arrowhead with
        :param point: The point to draw the arrowhead at, given in grid coords
        :type point: Tuple[float, float]
        """
        # This algorithm was adapted from a C# algorithm found at
        # http://csharphelper.com/blog/2014/12/draw-lines-with-arrowheads-in-c/

        # Get the x and y coords of the point, and then normalize them
        # We have to normalize them, or else the size of the arrowhead will
        # scale with the distance of the point from the origin
        x, y = point
        vector_length = np.sqrt(x * x + y * y)

        if vector_length < 1e-12:
```

```python
                return

        nx = x / vector_length
        ny = y / vector_length

        # We choose a length and find the steps in the x and y directions
        length = min(
            self.arrowhead_length * self.default_grid_spacing / self.grid_spacing,
            vector_length
        )
        dx = length * (-nx - ny)
        dy = length * (nx - ny)

        # Then we just plot those lines
        painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x + dx, y + dy))
        painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x - dy, y + dx))

    def draw_position_vector(self, painter: QPainter, point: Tuple[float, float], colour: QColor) -> None:
        """Draw a vector from the origin to the given point.

        :param QPainter painter: The painter to draw the position vector with
        :param point: The tip of the position vector in grid coords
        :type point: Tuple[float, float]
        :param QColor colour: The colour to draw the position vector in
        """
        painter.setPen(QPen(colour, self.width_vector_line))
        painter.drawLine(*self.canvas_origin, *self.canvas_coords(*point))
        self.draw_arrowhead_away_from_origin(painter, point)

    def draw_basis_vectors(self, painter: QPainter) -> None:
        """Draw arrowheads at the tips of the basis vectors.

        :param QPainter painter: The painter to draw the basis vectors with
        """
        self.draw_position_vector(painter, self.point_i, self.colour_i)
        self.draw_position_vector(painter, self.point_j, self.colour_j)

    def draw_determinant_parallelogram(self, painter: QPainter) -> None:
        """Draw the parallelogram of the determinant of the matrix.

        :param QPainter painter: The painter to draw the parallelogram with
        """
        if self.det == 0:
            return

        path = QPainterPath()
        path.moveTo(*self.canvas_origin)
        path.lineTo(*self.canvas_coords(*self.point_i))
        path.lineTo(*self.canvas_coords(self.point_i[0] + self.point_j[0], self.point_i[1] + self.point_j[1]))
        path.lineTo(*self.canvas_coords(*self.point_j))

        color = (16, 235, 253) if self.det > 0 else (253, 34, 16)
        brush = QBrush(QColor(*color, alpha=128), Qt.SolidPattern)

        painter.fillPath(path, brush)

    def draw_determinant_text(self, painter: QPainter) -> None:
        """Write the string value of the determinant in the middle of the parallelogram.

        :param QPainter painter: The painter to draw the determinant text with
        """
        painter.setPen(QPen(self.colour_text, self.width_vector_line))

        # We're building a QRect that encloses the determinant parallelogram
        # Then we can center the text in this QRect
        coords: List[Tuple[float, float]] = [
            (0, 0),
            self.point_i,
            self.point_j,
            (
                self.point_i[0] + self.point_j[0],
                self.point_i[1] + self.point_j[1]
            )
```

```
        ]

        xs = [t[0] for t in coords]
        ys = [t[1] for t in coords]

        top_left = QPoint(*self.canvas_coords(min(xs), max(ys)))
        bottom_right = QPoint(*self.canvas_coords(max(xs), min(ys)))

        rect = QRectF(top_left, bottom_right)

        painter.drawText(
            rect,
            Qt.AlignHCenter | Qt.AlignVCenter,
            f'{self.det:.2f}'
        )

    def draw_eigenvectors(self, painter: QPainter) -> None:
        """Draw the eigenvectors of the displayed matrix transformation.

        :param QPainter painter: The painter to draw the eigenvectors with
        """
        for value, vector in self.eigs:
            x = value * vector[0]
            y = value * vector[1]

            if x.imag != 0 or y.imag != 0:
                continue

            self.draw_position_vector(painter, (x, y), self.colour_eigen)

            # Now we need to draw the eigenvalue at the tip of the eigenvector

            offset = 3
            top_left: QPoint
            bottom_right: QPoint
            alignment_flags: int

            if x >= 0 and y >= 0:  # Q1
                top_left = QPoint(self.canvas_x(x) + offset, 0)
                bottom_right = QPoint(self.width(), self.canvas_y(y) - offset)
                alignment_flags = Qt.AlignLeft | Qt.AlignBottom

            elif x < 0 and y >= 0:  # Q2
                top_left = QPoint(0, 0)
                bottom_right = QPoint(self.canvas_x(x) - offset, self.canvas_y(y) - offset)
                alignment_flags = Qt.AlignRight | Qt.AlignBottom

            elif x < 0 and y < 0:  # Q3
                top_left = QPoint(0, self.canvas_y(y) + offset)
                bottom_right = QPoint(self.canvas_x(x) - offset, self.height())
                alignment_flags = Qt.AlignRight | Qt.AlignTop

            else:  # Q4
                top_left = QPoint(self.canvas_x(x) + offset, self.canvas_y(y) + offset)
                bottom_right = QPoint(self.width(), self.height())
                alignment_flags = Qt.AlignLeft | Qt.AlignTop

            painter.setPen(QPen(self.colour_text, self.width_vector_line))
            painter.drawText(QRectF(top_left, bottom_right), alignment_flags, f'{value:.2f}')

    def draw_eigenlines(self, painter: QPainter) -> None:
        """Draw the eigenlines (invariant lines).

        :param QPainter painter: The painter to draw the eigenlines with
        """
        painter.setPen(QPen(self.colour_eigen, self.width_transformed_grid))

        for value, vector in self.eigs:
            if value.imag != 0:
                continue

            x, y = vector

```

```
            if x == 0:
                x_mid = int(self.width() / 2)
                painter.drawLine(x_mid, 0, x_mid, self.height())

            elif y == 0:
                y_mid = int(self.height() / 2)
                painter.drawLine(0, y_mid, self.width(), y_mid)

            else:
                self.draw_oblique_line(painter, y / x, 0)
```

A.14 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

from sys import version_info
from typing import Any, List, Tuple

from numpy import ndarray
from nptyping import NDArray, Float

if version_info >= (3, 10):
    from typing import TypeGuard

__all__ = ['is_matrix_type', 'MatrixType', 'MatrixParseList', 'VectorType']

MatrixType = NDArray[(2, 2), Float]
"""This type represents a 2x2 matrix as a NumPy array."""

VectorType = NDArray[(2,), Float]
"""This type represents a 2D vector as a NumPy array, for use with :attr:`MatrixType`."""

MatrixParseList = List[List[Tuple[str, str, str]]]
"""This is a list containing lists of tuples. Each tuple represents a matrix and is ``(multiplier,
matrix_identifier, index)`` where all of them are strings. These matrix-representing tuples are
contained in lists which represent multiplication groups. Every matrix in the group should be
multiplied together, in order. These multiplication group lists are contained by a top level list,
which is this type. Once these multiplication group lists have been evaluated, they should be summed.

In the tuples, the multiplier is a string representing a real number, the matrix identifier
is a capital letter or ``rot(x)`` where x is a real number angle, and the index is a string
representing an integer, or it's the letter ``T`` for transpose.
"""


def is_matrix_type(matrix: Any) -> TypeGuard[NDArray[(2, 2), Float]]:
    """Check if the given value is a valid matrix type.

    .. note::
       This function is a TypeGuard, meaning if it returns True, then the
       passed value must be a :attr:`lintrans.typing_.MatrixType`.
    """
    return isinstance(matrix, ndarray) and matrix.shape == (2, 2)
```

A.15 `matrices/parse.py`

```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 functions to parse and validate matrix expressions."""

from __future__ import annotations

import re
from dataclasses import dataclass
from typing import List, Pattern, Tuple

from lintrans.typing_ import MatrixParseList

NAIVE_CHARACTER_CLASS = r'[-+\sA-Z0-9.rot()^{}]'


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()


def find_sub_expressions(expression: str) -> List[str]:
    """Find all the sub-expressions in the given expression.

    This function only goes one level deep, so may return strings like ``'A(BC)D'``.

    :raises MatrixParseError: If there are unbalanced parentheses
    """
    sub_expressions: List[str] = []
    string = ''
    paren_depth = 0
    pointer = 0

    while True:
        char = expression[pointer]

        if char == '(' and expression[pointer - 3:pointer] != 'rot':
            paren_depth += 1

            # This is a bit of a manual bodge, but it eliminates extraneous parens
            if paren_depth == 1:
                pointer += 1
                continue

        elif char == ')' and re.match(f'{NAIVE_CHARACTER_CLASS}*?rot\\([-\\d.]+$', expression[:pointer]) is None:
            paren_depth -= 1

        if paren_depth > 0:
```

```python
            string += char

        if paren_depth == 0 and string:
            sub_expressions.append(string)
            string = ''

        pointer += 1

        if pointer >= len(expression):
            break

    if paren_depth != 0:
        raise MatrixParseError('Unbalanced parentheses in expression')

    return sub_expressions


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

    This function simply checks the expression against the BNF schema documented in
    :ref:`expression-syntax-docs`. It is not aware of which matrices are actually defined
    in a wrapper. For an aware version of this function, use the
    :meth:`lintrans.matrices.wrapper.MatrixWrapper.is_valid_expression` method.

    :param str expression: The expression to be validated
    :returns bool: Whether the expression is valid according to the schema
    """
    # Remove all whitespace
    expression = re.sub(r'\s', '', expression)

    match = naive_expression_pattern.match(expression)

    if match is None:
        return False

    # Check that the whole expression was matched against
    if expression != match.group(0):
        return False

    try:
        sub_expressions = find_sub_expressions(expression)
    except MatrixParseError:
        return False

    if not sub_expressions:
        return True

    return all(validate_matrix_expression(m) for m in sub_expressions)


@dataclass
class MatrixToken:
    """A simple dataclass to hold information about a matrix token being parsed."""

    multiplier: str = ''
    identifier: str = ''
    exponent: str = ''

    @property
    def tuple(self) -> Tuple[str, str, str]:
        """Create a tuple of the token for parsing."""
        return self.multiplier, self.identifier, self.exponent


class ExpressionParser:
    """A class to hold state during parsing.

    Most of the methods in this class are class-internal and should not be used from outside.

    This class should be used like this:

    >>> ExpressionParser('3A^-1B').parse()
```

```python
        [[('3', 'A', '-1'), ('', 'B', '')]]
        >>> ExpressionParser('4(M^TA^2)^-2').parse()
        [[('4', 'M^{T}A^{2}', '-2')]]
        """

    def __init__(self, expression: str):
        """Create an instance of the parser with the given expression and initialise variables to use during parsing."""
        # Remove all whitespace
        expression = re.sub(r'\s', '', expression)

        # Check if it's valid
        if not validate_matrix_expression(expression):
            raise MatrixParseError('Invalid expression')

        # Wrap all exponents and transposition powers with {}
        expression = re.sub(r'(?<=\^)(-?\d+|T)(?=[^}]|$)', r'{\g<0>}', expression)

        # Remove any standalone minuses
        expression = re.sub(r'-(?=[A-Z])', '-1', expression)

        # Replace subtractions with additions
        expression = re.sub(r'-(?=\d+\.?\d*([A-Z]|rot))', '+-', expression)

        # Get rid of a potential leading + introduced by the last step
        expression = re.sub(r'^\+', '', expression)

        self.expression = expression
        self.pointer: int = 0

        self.current_token = MatrixToken()
        self.current_group: List[Tuple[str, str, str]] = []

        self.final_list: MatrixParseList = []

    def __repr__(self) -> str:
        """Return a simple repr containing the expression."""
        return f'{self.__class__.__module__}.{self.__class__.__name__}("{self.expression}")'

    @property
    def char(self) -> str:
        """Return the char pointed to by the pointer."""
        return self.expression[self.pointer]

    def parse(self) -> MatrixParseList:
        """Fully parse the instance's matrix expression and return the :attr:`lintrans.typing_.MatrixParseList`.

        This method uses all the private methods of this class to parse the
        expression in parts. All private methods mutate the instance variables.

        :returns: The parsed expression
        :rtype: :attr:`lintrans.typing_.MatrixParseList`
        """
        self._parse_multiplication_group()

        while self.pointer < len(self.expression):
            if self.expression[self.pointer] != '+':
                raise MatrixParseError('Expected "+" between multiplication groups')

            self.pointer += 1
            self._parse_multiplication_group()

        return self.final_list

    def _parse_multiplication_group(self) -> None:
        """Parse a group of matrices to be multiplied together.

        This method just parses matrices until we get to a ``+``.
        """
        # This loop continues to parse matrices until we fail to do so
        while self._parse_matrix():
            # Once we get to the end of the multiplication group, we add it the final list and reset the group list
            if self.pointer >= len(self.expression) or self.char == '+':
```

```python
                self.final_list.append(self.current_group)
                self.current_group = []
                self.pointer += 1

    def _parse_matrix(self) -> bool:
        """Parse a full matrix using :meth:`_parse_matrix_part`.

        This method will parse an optional multiplier, an identifier, and an optional exponent. If we
        do this successfully, we return True. If we fail to parse a matrix (maybe we've reached the
        end of the current multiplication group and the next char is ``+``), then we return False.

        :returns bool: Success or failure
        """
        self.current_token = MatrixToken()

        while self._parse_matrix_part():
            pass  # The actual execution is taken care of in the loop condition

        if self.current_token.identifier == '':
            return False

        self.current_group.append(self.current_token.tuple)
        return True

    def _parse_matrix_part(self) -> bool:
        """Parse part of a matrix (multiplier, identifier, or exponent).

        Which part of the matrix we parse is dependent on the current value of the pointer and the expression.
        This method will parse whichever part of matrix token that it can. If it can't parse a part of a matrix,
        or it's reached the next matrix, then we just return False. If we succeeded to parse a matrix part, then
        we return True.

        :returns bool: Success or failure
        :raises MatrixParseError: If we fail to parse this part of the matrix
        """
        if self.pointer >= len(self.expression):
            return False

        if self.char.isdigit() or self.char == '-':
            if self.current_token.multiplier != '' \
                    or (self.current_token.multiplier == '' and self.current_token.identifier != ''):
                return False

            self._parse_multiplier()

        elif self.char.isalpha() and self.char.isupper():
            if self.current_token.identifier != '':
                return False

            self.current_token.identifier = self.char
            self.pointer += 1

        elif self.char == 'r':
            if self.current_token.identifier != '':
                return False

            self._parse_rot_identifier()

        elif self.char == '(':
            if self.current_token.identifier != '':
                return False

            self._parse_sub_expression()

        elif self.char == '^':
            if self.current_token.exponent != '':
                return False

            self._parse_exponent()

        elif self.char == '+':
            return False

```

```python
        else:
            raise MatrixParseError(f'Unrecognised character "{self.char}" in matrix expression')

        return True

    def _parse_multiplier(self) -> None:
        """Parse a multiplier from the expression and pointer.

        This method just parses a numerical multiplier, which can include
        zero or one ``.`` character and optionally a ``-`` at the start.

        :raises MatrixParseError: If we fail to parse this part of the matrix
        """
        multiplier = ''

        while self.char.isdigit() or self.char in ('.', '-'):
            multiplier += self.char
            self.pointer += 1

        try:
            float(multiplier)
        except ValueError as e:
            raise MatrixParseError(f'Invalid multiplier "{multiplier}"') from e

        self.current_token.multiplier = multiplier

    def _parse_rot_identifier(self) -> None:
        """Parse a ``rot()``-style identifier from the expression and pointer.

        This method will just parse something like ``rot(12.5)``. The angle number must be a real number.

        :raises MatrixParseError: If we fail to parse this part of the matrix
        """
        if match := re.match(r'rot\(([\d.-]+)\)', self.expression[self.pointer:]):
            # Ensure that the number in brackets is a valid float
            try:
                float(match.group(1))
            except ValueError as e:
                raise MatrixParseError(f'Invalid angle number "{match.group(1)}" in rot-identifier') from e

            self.current_token.identifier = match.group(0)
            self.pointer += len(match.group(0))
        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

```

```python
        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 :data:`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: :data:`lintrans.typing_.MatrixParseList`
    """
    return ExpressionParser(expression).parse()
```

A.16 `matrices/__init__.py`

```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 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.17 matrices/utility.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>

"""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]:
    """Return the polar coordinates of a given (x, y) Cartesian coordinate.

    .. note:: We're returning the angle in the range [0, 2pi)
    """
    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
    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
```

```python
    :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.18 matrices/wrapper.py

```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 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, Tuple, Union

import numpy as np

from lintrans.typing_ import is_matrix_type, MatrixType
```

```python
from .parse import 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):
        """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:
```

```python
                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)))

        if name not in self._matrices:
            if validate_matrix_expression(name):
                return self.evaluate_expression(name)

            raise NameError(f'Unrecognised matrix name "{name}"')

        # We copy the matrix before we return it so the user can't accidentally mutate the matrix
        matrix = copy(self._matrices[name])

        if isinstance(matrix, str):
            return self.evaluate_expression(matrix)

        return matrix

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

        The new matrix may be a simple 2x2 NumPy array, or it could be a string, representing an
        expression in terms of other, previously defined matrices.

        :param str name: The name of the matrix to set the value of
        :param Optional[Union[MatrixType, str]] new_matrix: The value of the new matrix (could be None)

        :raises NameError: If the name isn't a legal matrix name
        :raises TypeError: If the matrix isn't a valid 2x2 NumPy array or expression in terms of other defined matrices
        :raises ValueError: If you attempt to define a matrix in terms of itself
        """
        if not (name in self._matrices and name != 'I'):
            raise NameError('Matrix name is illegal')

        if new_matrix is None:
            self._matrices[name] = None
            return

        if isinstance(new_matrix, str):
            if self.is_valid_expression(new_matrix):
                if name not in new_matrix:
                    self._matrices[name] = new_matrix
                    return
                else:
                    raise ValueError('Cannot define a matrix recursively')

        if not is_matrix_type(new_matrix):
            raise TypeError('Matrix must be a 2x2 NumPy array')

```

```
            # All matrices must have float entries
            a = float(new_matrix[0][0])
            b = float(new_matrix[0][1])
            c = float(new_matrix[1][0])
            d = float(new_matrix[1][1])

            self._matrices[name] = np.array([[a, b], [c, d]])

        def get_expression(self, name: str) -> Optional[str]:
            """If the named matrix is defined as an expression, return that expression, else return None.

            :param str name: The name of the matrix
            :returns Optional[str]: The expression that the matrix is defined as, or None

            :raises NameError: If the name is invalid
            """
            if name not in self._matrices:
                raise NameError('Matrix must have a legal name')

            matrix = self._matrices[name]
            if isinstance(matrix, str):
                return matrix

            return None

        def is_valid_expression(self, expression: str) -> bool:
            """Check if the given expression is valid, using the context of the wrapper.

            This method calls :func:`lintrans.matrices.parse.validate_matrix_expression`, but also
            ensures that all the matrices in the expression are defined in the wrapper.

            :param str expression: The expression to validate
            :returns bool: Whether the expression is valid in this wrapper

            :raises LinAlgError: If a matrix is defined in terms of the inverse of a singular matrix
            """
            # Get rid of the transposes to check all capital letters
            new_expression = expression.replace('^T', '').replace('^{T}', '')

            # Make sure all the referenced matrices are defined
            for matrix in [x for x in new_expression if re.match('[A-Z]', x)]:
                if self[matrix] is None:
                    return False

                if (expr := self.get_expression(matrix)) is not None:
                    if not self.is_valid_expression(expr):
                        return False

            return validate_matrix_expression(expression)

        def evaluate_expression(self, expression: str) -> MatrixType:
            """Evaluate a given expression and return the matrix evaluation.

            :param str expression: The expression to be parsed
            :returns MatrixType: The matrix result of the expression

            :raises ValueError: If the expression is invalid
            """
            if not self.is_valid_expression(expression):
                raise ValueError('The expression is invalid')

            parsed_result = parse_matrix_expression(expression)
            final_groups: List[List[MatrixType]] = []

            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
```

```python
                    assert m is not None
                    matrix_value = m.T

                else:
                    matrix_value = np.linalg.matrix_power(self[identifier], 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
```

B Testing code

B.1 matrices/test_parse_and_validate_expression.py

```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>

"""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, 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'

    '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^{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:
```

```python
    """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', '')]]),
    ('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


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)
```

B.2 matrices/utility/test_rotation_matrices.py

```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>

"""Test functions for rotation matrices."""

from typing import List, Tuple

import numpy as np
import pytest

from lintrans.matrices import create_rotation_matrix
from lintrans.typing_ import MatrixType

angles_and_matrices: List[Tuple[float, float, MatrixType]] = [
    (0, 0, np.array([[1, 0], [0, 1]])),
    (90, np.pi / 2, np.array([[0, -1], [1, 0]])),
    (180, np.pi, np.array([[-1, 0], [0, -1]])),
    (270, 3 * np.pi / 2, np.array([[0, 1], [-1, 0]])),
    (360, 2 * np.pi, np.array([[1, 0], [0, 1]])),

    (45, np.pi / 4, np.array([
        [np.sqrt(2) / 2, -1 * np.sqrt(2) / 2],
        [np.sqrt(2) / 2, np.sqrt(2) / 2]
    ])),
    (135, 3 * np.pi / 4, np.array([
        [-1 * np.sqrt(2) / 2, -1 * np.sqrt(2) / 2],
        [np.sqrt(2) / 2, -1 * np.sqrt(2) / 2]
    ])),
    (225, 5 * np.pi / 4, np.array([
        [-1 * np.sqrt(2) / 2, np.sqrt(2) / 2],
        [-1 * np.sqrt(2) / 2, -1 * np.sqrt(2) / 2]
    ])),
    (315, 7 * np.pi / 4, np.array([
        [np.sqrt(2) / 2, np.sqrt(2) / 2],
        [-1 * np.sqrt(2) / 2, np.sqrt(2) / 2]
    ])),

    (30, np.pi / 6, np.array([
        [np.sqrt(3) / 2, -1 / 2],
        [1 / 2, np.sqrt(3) / 2]
    ])),
    (60, np.pi / 3, np.array([
        [1 / 2, -1 * np.sqrt(3) / 2],
        [np.sqrt(3) / 2, 1 / 2]
    ])),
    (120, 2 * np.pi / 3, np.array([
        [-1 / 2, -1 * np.sqrt(3) / 2],
        [np.sqrt(3) / 2, -1 / 2]
    ])),
    (150, 5 * np.pi / 6, np.array([
        [-1 * np.sqrt(3) / 2, -1 / 2],
        [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."""
```

```python
    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()
```

B.3 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.4 matrices/utility/test_float_utility_functions.py

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

```python

# 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


def test_round_float() -> None:
    """Test the round_float() function."""
    expected_values: List[Tuple[float, int, str]] = [
        (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'),
        (np.sqrt(2) / 2, 5, '0.70711'), (-1 * np.sqrt(2) / 2, 5, '-0.70711'),
        (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'),
        (1.23456789, 2, '1.23'), (1.23456789, 3, '1.235'), (1.23456789, 4, '1.2346'), (1.23456789, 5, '1.23457'),
        (12345.678, 1, '12345.7'), (12345.678, 2, '12345.68'), (12345.678, 3, '12345.678'),
    ]

    for num, precision, answer in expected_values:
        assert round_float(num, precision) == answer
```

B.5 matrices/matrix_wrapper/test_misc.py

```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>

"""Test the miscellaneous methods of the MatrixWrapper class."""

from lintrans.matrices import MatrixWrapper


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
```

```python
    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-40'
    assert test_wrapper.get_expression('S') == 'NOP'
```

B.6 matrices/matrix_wrapper/test_setitem_and_getitem.py

```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>

"""Test the MatrixWrapper __setitem__() and __getitem__() methods."""

from typing import Any, 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'


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'
    test_wrapper['P'] = 'A+C'

    test_wrapper['Q'] = 'N^-1'
    test_wrapper['R'] = 'P-4O'
    test_wrapper['S'] = 'NOP'
```

```python
    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_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


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
```

B.7 matrices/matrix_wrapper/conftest.py

```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>

"""A simple conftest.py containing some re-usable fixtures."""

import numpy as np
import pytest

from lintrans.matrices import MatrixWrapper


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]])

```

```python
        return wrapper


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


@pytest.fixture
def new_wrapper() -> MatrixWrapper:
    """Return a new MatrixWrapper with no initialized values."""
    return MatrixWrapper()
```

B.8 matrices/matrix_wrapper/test_evaluate_expression.py

```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>

"""Test the MatrixWrapper evaluate_expression() method."""

import numpy as np
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()
```

```python

    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()

    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."""
    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}') == test_wrapper['A'].T).all()
    assert (test_wrapper.evaluate_expression('B^{T}') == test_wrapper['B'].T).all()
    assert (test_wrapper.evaluate_expression('C^{T}') == test_wrapper['C'].T).all()
    assert (test_wrapper.evaluate_expression('D^{T}') == test_wrapper['D'].T).all()
    assert (test_wrapper.evaluate_expression('E^{T}') == test_wrapper['E'].T).all()
    assert (test_wrapper.evaluate_expression('F^{T}') == test_wrapper['F'].T).all()
    assert (test_wrapper.evaluate_expression('G^{T}') == test_wrapper['G'].T).all()

    assert (test_wrapper.evaluate_expression('A^T') == test_wrapper['A'].T).all()
    assert (test_wrapper.evaluate_expression('B^T') == test_wrapper['B'].T).all()
    assert (test_wrapper.evaluate_expression('C^T') == test_wrapper['C'].T).all()
    assert (test_wrapper.evaluate_expression('D^T') == test_wrapper['D'].T).all()
    assert (test_wrapper.evaluate_expression('E^T') == test_wrapper['E'].T).all()
    assert (test_wrapper.evaluate_expression('F^T') == test_wrapper['F'].T).all()
    assert (test_wrapper.evaluate_expression('G^T') == test_wrapper['G'].T).all()

    assert test_wrapper == get_test_wrapper()


def test_rotation_matrices(test_wrapper: MatrixWrapper) -> None:
    """Test that 'rot(angle)' can be used in an expression."""
    assert (test_wrapper.evaluate_expression('rot(90)') == create_rotation_matrix(90)).all()
    assert (test_wrapper.evaluate_expression('rot(180)') == create_rotation_matrix(180)).all()
    assert (test_wrapper.evaluate_expression('rot(270)') == create_rotation_matrix(270)).all()
    assert (test_wrapper.evaluate_expression('rot(360)') == create_rotation_matrix(360)).all()
    assert (test_wrapper.evaluate_expression('rot(45)') == create_rotation_matrix(45)).all()
    assert (test_wrapper.evaluate_expression('rot(30)') == create_rotation_matrix(30)).all()

    assert (test_wrapper.evaluate_expression('rot(13.43)') == create_rotation_matrix(13.43)).all()
    assert (test_wrapper.evaluate_expression('rot(49.4)') == create_rotation_matrix(49.4)).all()
    assert (test_wrapper.evaluate_expression('rot(-123.456)') == create_rotation_matrix(-123.456)).all()
    assert (test_wrapper.evaluate_expression('rot(963.245)') == create_rotation_matrix(963.245)).all()
    assert (test_wrapper.evaluate_expression('rot(-235.24)') == create_rotation_matrix(-235.24)).all()

    assert test_wrapper == get_test_wrapper()


def test_multiplication_and_addition(test_wrapper: MatrixWrapper) -> None:
    """Test multiplication and addition of matrices together."""
    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 \
```

```python
           test_wrapper['G'] is not None

    assert (test_wrapper.evaluate_expression('AB+C') ==
            test_wrapper['A'] @ test_wrapper['B'] + test_wrapper['C']).all()
    assert (test_wrapper.evaluate_expression('DE-D') ==
            test_wrapper['D'] @ test_wrapper['E'] - test_wrapper['D']).all()
    assert (test_wrapper.evaluate_expression('FD+AB') ==
            test_wrapper['F'] @ test_wrapper['D'] + test_wrapper['A'] @ test_wrapper['B']).all()
    assert (test_wrapper.evaluate_expression('BA-DE') ==
            test_wrapper['B'] @ test_wrapper['A'] - test_wrapper['D'] @ test_wrapper['E']).all()

    assert (test_wrapper.evaluate_expression('2AB+3C') ==
            (2 * test_wrapper['A']) @ test_wrapper['B'] + (3 * test_wrapper['C'])).all()
    assert (test_wrapper.evaluate_expression('4D7.9E-1.2A') ==
            (4 * test_wrapper['D']) @ (7.9 * test_wrapper['E']) - (1.2 * test_wrapper['A'])).all()

    assert test_wrapper == get_test_wrapper()


def test_complicated_expressions(test_wrapper: MatrixWrapper) -> None:
    """Test evaluation of complicated 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('-3.2A^T 4B^{-1} 6C^{-1} + 8.1D^{2} 3.2E^4') ==
            (-3.2 * test_wrapper['A'].T) @ (4 * la.inv(test_wrapper['B'])) @ (6 * la.inv(test_wrapper['C']))
            + (8.1 * la.matrix_power(test_wrapper['D'], 2)) @ (3.2 * la.matrix_power(test_wrapper['E'], 4))).all()

    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
```

```python
                ),
                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()
```