`lintrans`

by Dyson 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[15], 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. Multiple 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. They also said that an online version would probably be easier to use, but I have absolutely no experience in web development and I'm much more comfortable making a desktop app.

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

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

They also agreed that a tutorial on using the software would be useful, so I plan to implement this through an online written tutorial hosted with GitHub Pages, and perhaps a video tutorial as well. This would make the app much easier to use for people who have never seen it before. It wouldn't be a lesson on the maths itself, but 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'[16] 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.

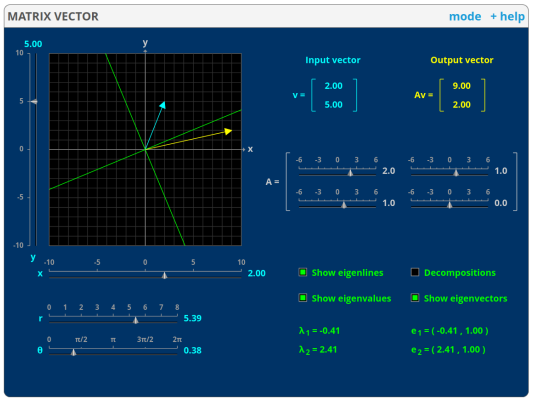


Figure 1.1: The MIT 'Matrix Vector' Mathlet

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

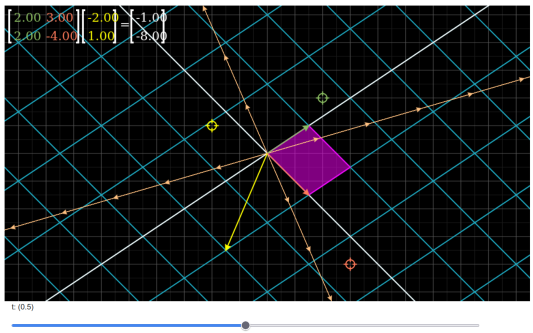


Figure 1.2: 'Linear Transformation Visualizer' halfway through an animation

Another web app that I found was one simply called 'Linear Transformation Visualizer' by Shad Sharma[40]. 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.

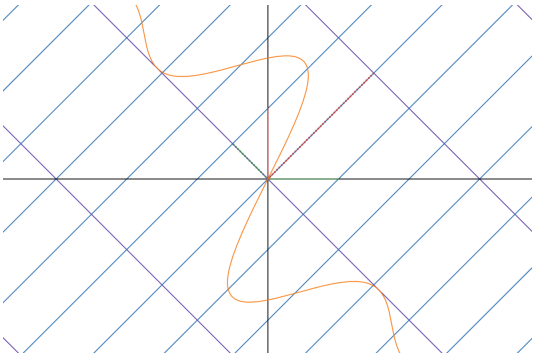


Figure 1.3: The Desmos app halfway through an animation, rendering $f(x) = \frac{\sin^2 x}{x}$ in orange

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

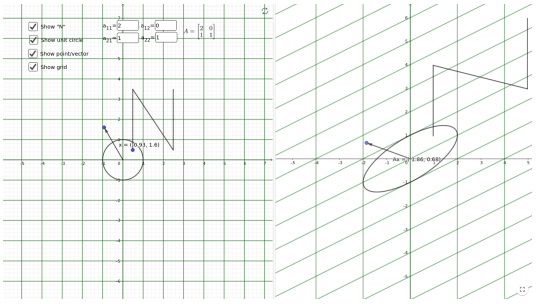


Figure 1.4: The GeoGebra applet rendering its default matrix

The last solution that I want to talk about is a GeoGebra applet simply titled 'Visualizing Linear Transformations'[18]. This applet has input and output vectors, original and transformed grid lines, a unit circle, and the letter N. It allows the user to define a matrix as 4 numbers and view the aforementioned N (which the user can translate to anywhere on the grid), the unit circle, the input/output vectors, and the grid lines. It also has the input vector snapping to integer coordinates, but that's a standard part of GeoGebra.

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

1.4 Essential features

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

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

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

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

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

1.5 Limitations

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

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

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

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

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

1.6 Hardware and software requirements

1.6.1 Hardware

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

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

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

1.6.2 Software

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

[1] Python 3.8 or higher won't compile on any earlier versions of macOS[30]

[2] Specifying a Linux version is practically impossible. Python 3.8 or higher is available in many package repositories, but all modern Python versions will compile on any modern distro. Qt5 is available in many package repositories and can be compiled on any `x86` or `x86_64` generic Linux machine with `gcc` version 5 or later[31]

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

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

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

1.7 Success criteria

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

Additionally, the app must fulfil some basic requirements:

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

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

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

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

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

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

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

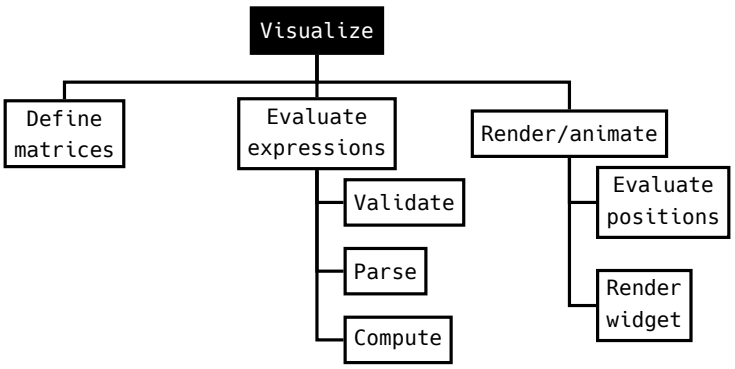
Transforming polygons is the lowest priority item on this list and will likely be implemented last, but it would definitely benefit learning. I wouldn't consider it necessary for success, but it would be very good to include, and it's certainly a feature that I want to have.

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

2 Design

2.1 Problem decomposition

I have decomposed the problem of visualization as follows:



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

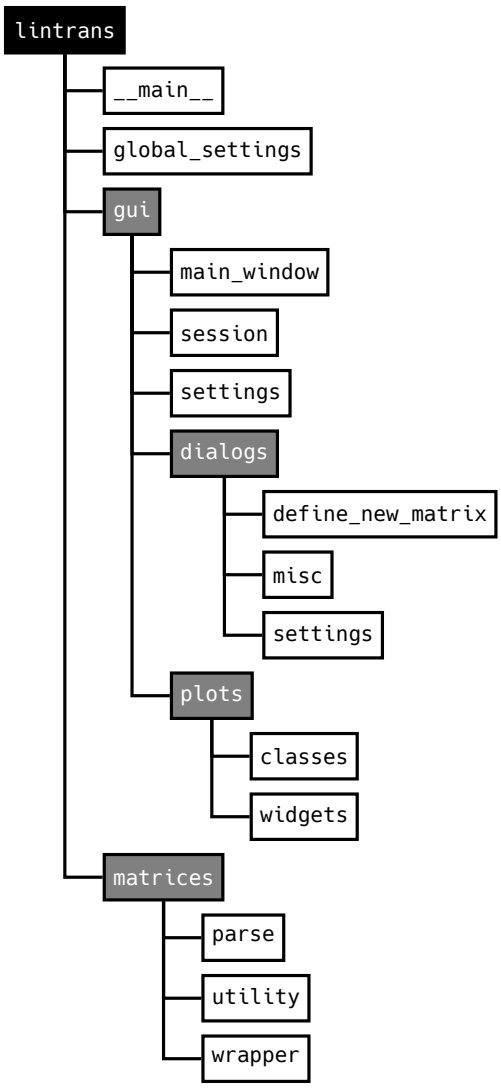
Evaluating expressions is another key part of the app and can be further broken down into validating, parsing, and computing the value. Validating an expression simply consists of checking that it adheres to a set of syntax rules for matrix expressions, and that it only contains matrices which have already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

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

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

2.2 Structure of the solution

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, using this module as the entry point.

The `global_settings` module will define a `GlobalSettings` singleton class. This class will manage global settings and variables - things like where to save sessions by default, etc. I'm not entirely sure what I want to put in here, but I expect that I'll want global settings in the future. Having this class will allow me to easily read and write these settings to a file to have them persist between sessions.

`matrices` is the package that will allow the user to define, validate, parse, evaluate, and use matrices. The `matrices.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 in the `matrices.wrapper` module. This `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 the matrices used in an expression are actually defined in the wrapper - as well the ability to evaluate matrix expressions, in addition to its basic behaviour of setting and getting matrices by name. There will also be a `matrices.utility` module, which will contain some simple functions for simple functionality. Functions like `create_rotation_matrix()`, which will generate a rotation matrix from an angle using the formula $\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}$.

`gui` is the package that will contain all the frontend code for everything GUI-related. `gui.main_window` is the module that will define the `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. The most important part of the main window is the viewport, which will be discussed shortly. This module will also contain a simple `main()` function to instantiate and launch the application GUI.

The `gui.session` module will contain functions to save and load a session from a file. A session will consist of the `MatrixWrapper`, along with perhaps the display settings and maybe some other things. I know that saving the wrapper will be essential, but I'll see what else should be saved as the project evolves.

The `gui.settings` module will contain a `DisplaySettings` dataclass[5] that will represent the settings for visualizing transformations. The viewport 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 `gui.dialogs` subpackage will contain modules with different dialog classes. It will have a `gui.dialogs.define_new_matrices` module, which will have a `DefineDialog` abstract superclass. It will then contain classes that inherit from this superclass and provide dialogs for defining new matrices visually,

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

numerically, and as an expression in terms of other matrices. Additionally, it will contain a `gui.dialogs.settings` module, which will provide a `SettingsDialog` superclass and a `DisplaySettingsDialog` class, which will allow the user to configure the aforementioned display settings. It may also have a `GlobalSettingsDialog` class in the future, which would similarly allow the user to configure the app's global settings through a dialog. This will only be implemented once I've actually got global settings to configure.

The `gui.dialogs.misc` module will contain small miscellaneous dialog boxes - things like the about box which are very simple and don't need a dedicated module.

The `gui.plots` subpackage will have a `gui.plots.classes` module and a `gui.plots.widgets` module. The `classes` module will have the abstract superclasses `BackgroundPlot` and `VectorGridPlot`. The former will provide helper 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 `gui.plots.widgets` module will have the classes `VisualizeTransformationWidget` and `DefineVisuallyWidget`, which will both 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.

I also want the user to be able to define arbitrary polygons and view their transformations. I imagine this polygon definition will happen in a separate dialog, but I don't know where that's going to fit just yet. I'll probably have the widget in `gui.plots.widgets`, but possibly elsewhere.

2.3 Algorithm design

The project will have many algorithms but a lot of them will be related to drawing transformations on the canvas itself, and almost all of the algorithms will evolve over time. In this section, I will present pseudocode for some of the most interesting parts of the project.

The `lintrans.matrices.utility` module will look roughly like this:

```
import numpy as np // Python import

// Create a matrix representing a rotation (anticlockwise) by the given angle
function create_rotation_matrix(angle: float, degrees: bool = True) -> MatrixType
    if (degrees) then
        rad = np.deg2rad(angle MOD 360)
    else
        rad = angle MOD (2 * np.pi)
    endif

    return np.array([
        [np.cos(rad), -1 * np.sin(rad)],
        [np.sin(rad), np.cos(rad)]
    ])
endfunction
```

And the `lintrans.matrices.wrapper` module will look like this:

```
import re
import numpy as np

// The `.utility` syntax means that the utility module is next to this one in the tree
from .utility import create_rotation_matrix

```

```
class MatrixWrapper    // This is a hashmap from string to matrices, but the matrices might be null
    private matrices: Dict[string, Optional[MatrixType]]

    public procedure new()
        matrices = {
            "A": null, "B": null, "C": null, "D": null,
            "E": null, "F": null, "G": null, "H": null,
            "I": np.eye(2),  // I is always defined as the identity matrix
            "J": null, "K": null, "L": null, "M": null,
            "N": null, "O": null, "P": null, "Q": null,
            "R": null, "S": null, "T": null, "U": null,
            "V": null, "W": null, "X": null, "Y": null,
            "Z": null
        }
    endprocedure

    // This is a Python "magic method", which enable syntax like `wrapper['A']` to get the matrix A
    public function getitem(name: string) -> Optional[MatrixType]
        // 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

        // Return a new rotation matrix
        match = re.match(r"^rot\((-?\d*\.?\d*)\)$", name)
        if (match != null)
            return create_rotation_matrix(float(match.group(1)))
        endif

        if (!matrices.contains(name))
            raise NameError(f'Unrecognised matrix name "{name}"')
        endif

        return matrices[name]
    endfunction

    // Again, this is Python magic. This one allows assignments like `wrapper['A'] = my_matrix`
    public procedure setitem(self, name: string, new_matrix: Optional[MatrixType])
        // If new_matrix is null, then that effectively unsets the matrix name.

        if (name == "I" OR !matrices.contains(name))
            raise NameError("Matrix name is illegal")
        endif

        if (new_matrix == null)
            matrices[name] = null
            return
        endif

        if (!is_matrix_type(new_matrix))
            raise TypeError("Matrix must be a 2x2 NumPy array")
        endif

        // 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])

        matrices[name] = np.array([[a, b], [c, d]])
    endprocedure
endclass
```

These modules handle the creation, storage, and use of matrices. Their implementations are deliberately simple, since they don't have to do much. I will eventually extend the `MatrixWrapper` class to allow strings as matrices, so they can be defined as expressions, but this is unnecessary for now. It will simply be more conditions in `__getitem__` and `__setitem__` and a method to evaluate expressions.

Parsing matrix expressions will be quite tricky and I don't really know how I'm going to do it. I think it will be possible with regular expressions, since I won't support nested expressions at first. But adding support for nested expressions may require something more complicated. I will have a function to validate a matrix expression, which can definitely be done with regular expressions, and

I'll have another public function to parse matrix expressions, although this one may use some private functions to implement it properly.

I'm not sure on any algorithms yet, but here's the full BNF specification for matrix expressions (including nested expressions):

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

Obviously the data structure returned by the parser is very important. I have decided to use `list[list[tuple[str, str, str]]]`. Each tuple represents a real number multiplier, a matrix identifier, and an index. The multiplier and index may be empty strings. These tuples are contained in lists which represent matrices to be multiplied together, in order. Those lists are contained in a top level list, which represents multiplication groups which should be summed.

This type makes the structure of the input clear, and makes it very easy for the `MatrixWrapper` to evaluate a parsed expression.

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[45], 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[20] 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.

[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

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

The most important variables in the project will be instance attributes on the `LintransMainWindow` class. It will have a `MatrixWrapper` instance, a `DisplaySettings` instance, and most importantly, a `VisualizeTransformationWidget` 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 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

[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()`

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`, which is needed to verify input when defining a matrix visually.

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^{4}5.6rot(19.2^T-B^{-1}4.1C^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 `MatrixParseError`, 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[29] to allow indexing into a `MatrixWrapper` object like `wrapper['M']`. This simplifies using the class.

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

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

import numpy as np

from lintrans.typing import MatrixType


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

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

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

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

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

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

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

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

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

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

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

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

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

"""Test the MatrixWrapper class."""

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

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


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


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

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


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


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


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


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

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

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

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

3.1.2 Rudimentary parsing and evaluating

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

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

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

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

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

11  class MatrixWrapper:
...
27      def __getitem__(self, name: str) -> Optional[MatrixType]:
28          """Get the matrix with the given name.
29
30          If it is a simple name, it will just be fetched from the dictionary.
31          If the name is followed with a 't', then we will return the transpose of the named matrix.
32          If the name is 'rot()', with a given angle in degrees, then we return a new rotation matrix with that angle.
33
34          :param str name: The name of the matrix to get
35          :returns: The value of the matrix (may be none)
36          :rtype: Optional[MatrixType]
```

```
37
38          :raises NameError: If there is no matrix with the given name
39          """
40          # Return a new rotation matrix
41          match = re.match(r'rot\((\d+)\)', name)
42          if match is not None:
43              return create_rotation_matrix(float(match.group(1)))
44
45          # Return the transpose of this matrix
46          match = re.match(r'([A-Z])t', name)
47          if match is not None:
48              matrix = self[match.group(1)]
49
50              if matrix is not None:
51                  return matrix.T
52              else:
53                  return None
54
55          if name not in self._matrices:
56              raise NameError(f'Unrecognised matrix name "{name}"')
57
58          return self._matrices[name]
59
```

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

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

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

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

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

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

11  class MatrixWrapper:
...
83      def parse_expression(self, expression: str) -> MatrixType:
84          """Parse a given expression and return the matrix for that expression.
85
86          Expressions are written with standard LaTeX notation for exponents. All whitespace is ignored.
87
88          Here is documentation on syntax:
89              A single matrix is written as 'A'.
90              Matrix A multiplied by matrix B is written as 'AB'
91              Matrix A plus matrix B is written as 'A+B'
92              Matrix A minus matrix B is written as 'A-B'
93              Matrix A squared is written as 'A^2'
94              Matrix A to the power of 10 is written as 'A^10' or 'A^{10}'
95              The inverse of matrix A is written as 'A^-1' or 'A^{-1}'
```

```
96              The transpose of matrix A is written as 'A^T' or 'At'
97
98          :param str expression: The expression to be parsed
99          :returns MatrixType: The matrix result of the expression
100
101         :raises ValueError: If the expression is invalid, such as an empty string
102         """
103         if expression == '':
104             raise ValueError('The expression cannot be an empty string')
105
106         match = re.search(r'[^-+A-Z^{}rot()\d.]', expression)
107         if match is not None:
108             raise ValueError(f'Invalid character "{match.group(0)}"')
109
110         # Remove all whitespace in the expression
111         expression = re.sub(r'\s', '', expression)
112
113         # Wrap all exponents and transposition powers with {}
114         expression = re.sub(r'(?<=\^)(-?\d+|T)(?=[^}]|$)', r'{\g<0>}', expression)
115
116         # Replace all subtractions with additions, multiplied by -1
117         expression = re.sub(r'(?<=.)-(?=[A-Z])', '+-1', expression)
118
119         # Replace a possible leading minus sign with -1
120         expression = re.sub(r'^-(?=[A-Z])', '-1', expression)
121
122         # Change all transposition exponents into lowercase
123         expression = expression.replace('^{T}', 't')
124
125         # Split the expression into groups to be multiplied, and then we add those groups at the end
126         # We also have to filter out the empty strings to reduce errors
127         multiplication_groups = [x for x in expression.split('+') if x != '']
128
129         # Start with the 0 matrix and add each group on
130         matrix_sum: MatrixType = np.array([[0., 0.], [0., 0.]])
131
132         for group in multiplication_groups:
133             # Generate a list of tuples, each representing a matrix
134             # These tuples are (the multiplier, the matrix (with optional
135             # 't' at the end to indicate a transpose), the exponent)
136             string_matrices: list[tuple[str, str, str]]
137
138             # The generate tuple is (multiplier, matrix, full exponent, stripped exponent)
139             # The full exponent contains ^{}, so we ignore it
140             # The multiplier and exponent might be '', so we have to set them to '1'
141             string_matrices = [(t[0] if t[0] != '' else '1', t[1], t[3] if t[3] != '' else '1')
142                                for t in re.findall(r'(-?\d*\.?\d*)([A-Z]t?|rot\(\d+\))(\^{(-?\d+|T)})?', group)]
143
144             # This list is a list of tuple, where each tuple is (a float multiplier,
145             # the matrix (gotten from the wrapper's __getitem__()), the integer power)
146             matrices: list[tuple[float, MatrixType, int]]
147             matrices = [(float(t[0]), self[t[1]], int(t[2])) for t in string_matrices]
148
149             # Process the matrices and make actual MatrixType objects
150             processed_matrices: list[MatrixType] = [t[0] * np.linalg.matrix_power(t[1], t[2]) for t in matrices]
151
152             # Add this matrix product to the sum total
153             matrix_sum += reduce(lambda m, n: m @ n, processed_matrices)
154
155         return matrix_sum
```

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

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

I then implemented several tests for this parsing.

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

"""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()
    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

class MatrixWrapper:
...
    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:

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

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

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

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

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

```
32      def compile_valid_expression_pattern() -> Pattern[str]:
33          """Compile the single regular expression that will match a valid matrix expression."""
34          digit_no_zero = '[123456789]'
35          digits = '\\d+'
36          integer_no_zero = digit_no_zero + '(' + digits + ')?'
37          real_number = f'({integer_no_zero}(\\.{digits})?|0?\\.{digits})'
38
39          index_content = f'(-?{integer_no_zero}|T)'
40          index = f'(\\^\\{{{index_content}\\}}|\\^{index_content}|t)'
41          matrix_identifier = f'([A-Z]|rot\\(-?{real_number}\\))'
42          matrix = '(' + real_number + '?' + matrix_identifier + index + '?)'
43          expression = f'-?{matrix}+((\\+|-){matrix}+)*'
44
45          return re.compile(expression)
```

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

Compare the old version:

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

29      def validate_matrix_expression(expression: str) -> bool:
...
36          Here is the schema for a valid expression given in a version of BNF:
...
38              expression        ::=  matrices { ( "+" | "-" ) matrices };
39              matrices          ::=  matrix { matrix };
40              matrix            ::=  [ real_number ] matrix_identifier [ index ];
41              matrix_identifier ::=  "A" .. "Z" | "rot(" real_number ")";
42              index             ::=  "^{" index_content "}" | "^" index_content | "t";
43              index_content     ::=  integer_not_zero | "T";
44
45              digit_no_zero     ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
46              digit             ::=  "0" | digit_no_zero;
47              digits            ::=  digit | digits digit;
48              integer_not_zero  ::=  [ "-" ] digit_no_zero [ digits ];
49              real_number       ::=  ( integer_not_zero [ "." digits ] | [ "-" ] [ "0" ] "." digits );
```

to the new version:

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

52      def validate_matrix_expression(expression: str) -> bool:
...
59          Here is the schema for a valid expression given in a version of BNF:
...
61              expression        ::=  [ "-" ] matrices { ( "+" | "-" ) matrices };
62              matrices          ::=  matrix { matrix };
63              matrix            ::=  [ real_number ] matrix_identifier [ index ];
64              matrix_identifier ::=  "A" .. "Z" | "rot(" [ "-" ] real_number ")";
65              index             ::=  "^{" index_content "}" | "^" index_content | "t";
66              index_content     ::=  [ "-" ] integer_not_zero | "T";
67
68              digit_no_zero     ::=  "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9";
69              digit             ::=  "0" | digit_no_zero;
70              digits            ::=  digit | digits digit;
71              integer_not_zero  ::=  digit_no_zero [ digits ];
72              real_number       ::=  ( integer_not_zero [ "." digits ] | [ "0" ] "." digits );
```

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

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

```python
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.

    :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

class MatrixWrapper:
...
    def evaluate_expression(self, expression: str) -> MatrixType:
...
        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] = []

```

```
177             for matrix in group:
178                 if matrix[2] == 'T':
179                     m = self[matrix[1]]
180                     assert m is not None
181                     matrix_value = m.T
182                 else:
183                     matrix_value = np.linalg.matrix_power(self[matrix[1]],
184                                                           1 if (index := matrix[2]) == '' else int(index))
185 
186                 matrix_value *= 1 if (multiplier := matrix[0]) == '' else float(multiplier)
187                 f_group.append(matrix_value)
188 
189             final_groups.append(f_group)
190 
191         return reduce(add, [reduce(matmul, group) for group in final_groups])
```

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

3.2 Initial GUI

3.2.1 First basic GUI

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

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

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

import sys

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

from lintrans.matrices import MatrixWrapper


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

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

        self.matrix_wrapper = MatrixWrapper()

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

        # === Create widgets

        # Left layout: the plot and input box

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

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

        # Right layout: all the buttons

        # Misc buttons

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

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

        # Define new matrix buttons

        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')
```

```
60              self.button_define_visually.setToolTip('Drag the basis vectors')
61
62              self.button_define_numerically = QtWidgets.QPushButton(self)
63              self.button_define_numerically.setText('Numerically')
64              self.button_define_numerically.setToolTip('Define a matrix just with numbers')
65
66              self.button_define_as_rotation = QtWidgets.QPushButton(self)
67              self.button_define_as_rotation.setText('As a rotation')
68              self.button_define_as_rotation.setToolTip('Define an angle to rotate by')
69
70              self.button_define_as_expression = QtWidgets.QPushButton(self)
71              self.button_define_as_expression.setText('As an expression')
72              self.button_define_as_expression.setToolTip('Define a matrix in terms of other matrices')
73
74              # Render buttons
75
76              self.button_render = QtWidgets.QPushButton(self)
77              self.button_render.setText('Render')
78              self.button_render.setEnabled(False)
79              self.button_render.clicked.connect(self.render_expression)
80              self.button_render.setToolTip('Render the expression<br><b>(Ctrl + Enter)</b>')
81
82              self.button_render_shortcut = QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Return'), self)
83              self.button_render_shortcut.activated.connect(self.button_render.click)
84
85              self.button_animate = QtWidgets.QPushButton(self)
86              self.button_animate.setText('Animate')
87              self.button_animate.setEnabled(False)
88              self.button_animate.clicked.connect(self.animate_expression)
89              self.button_animate.setToolTip('Animate the expression<br><b>(Ctrl + Shift + Enter)</b>')
90
91              self.button_animate_shortcut = QtWidgets.QShortcut(QtGui.QKeySequence('Ctrl+Shift+Return'), self)
92              self.button_animate_shortcut.activated.connect(self.button_animate.click)
93
94              # === Arrange widgets
95
96              self.setContentsMargins(10, 10, 10, 10)
97
98              self.vlay_left = QVBoxLayout()
99              self.vlay_left.addWidget(self.plot)
100             self.vlay_left.addWidget(self.text_input_expression)
101
102             self.vlay_misc_buttons = QVBoxLayout()
103             self.vlay_misc_buttons.setSpacing(20)
104             self.vlay_misc_buttons.addWidget(self.button_create_polygon)
105             self.vlay_misc_buttons.addWidget(self.button_change_display_settings)
106
107             self.vlay_define_new_matrix = QVBoxLayout()
108             self.vlay_define_new_matrix.setSpacing(20)
109             self.vlay_define_new_matrix.addWidget(self.label_define_new_matrix)
110             self.vlay_define_new_matrix.addWidget(self.button_define_visually)
111             self.vlay_define_new_matrix.addWidget(self.button_define_numerically)
112             self.vlay_define_new_matrix.addWidget(self.button_define_as_rotation)
113             self.vlay_define_new_matrix.addWidget(self.button_define_as_expression)
114
115             self.vlay_render = QVBoxLayout()
116             self.vlay_render.setSpacing(20)
117             self.vlay_render.addWidget(self.button_animate)
118             self.vlay_render.addWidget(self.button_render)
119
120             self.vlay_right = QVBoxLayout()
121             self.vlay_right.setSpacing(50)
122             self.vlay_right.addLayout(self.vlay_misc_buttons)
123             self.vlay_right.addLayout(self.vlay_define_new_matrix)
124             self.vlay_right.addLayout(self.vlay_render)
125
126             self.hlay_all = QHBoxLayout()
127             self.hlay_all.setSpacing(15)
128             self.hlay_all.addLayout(self.vlay_left)
129             self.hlay_all.addLayout(self.vlay_right)
130
131             self.central_widget = QtWidgets.QWidget()
132             self.central_widget.setLayout(self.hlay_all)
```

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

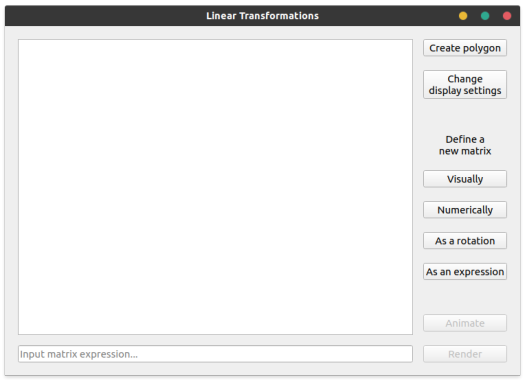


Figure 3.1: The first version of the GUI

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

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

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

3.2.2 Numerical definition dialog

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

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

1   """The module to provide dialogs for defining new matrices."""
2
3   from numpy import array
4   from PyQt5 import QtGui, QtWidgets
5   from PyQt5.QtWidgets import QDialog, QGridLayout, QHBoxLayout, QVBoxLayout
6
7   from lintrans.matrices import MatrixWrapper
8
9   ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
10
11
12  def is_float(string: str) -> bool:
13      """Check if a string is a float."""
```

```
        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()
        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)
```

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

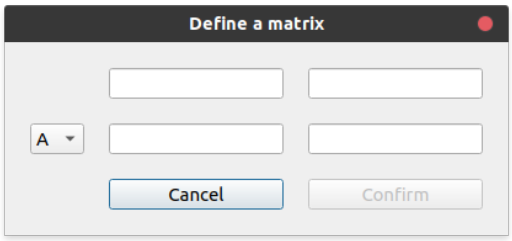


Figure 3.2: The first version of the numerical definition dialog

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

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

The `confirm_matrix()` method just updates the instance's matrix wrapper with the new matrix. We

pass a reference to the `LintransMainWindow` instance's matrix wrapper when we open the dialog, so we're just updating the referenced object directly.

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

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

12  class LintransMainWindow(QMainWindow):
...
15      def __init__(self):
...
66          self.button_define_numerically.clicked.connect(
67              lambda: DefineNumericallyDialog(self.matrix_wrapper, self).exec()
68          )
```

3.2.3 More definition dialogs

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

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

22  class DefineDialog(QDialog):
23      """A superclass for definitions dialogs."""
24
25      def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
26          """Create the dialog, but don't run it yet.
27
28          :param matrix_wrapper: The MatrixWrapper that this dialog will mutate
29          :type matrix_wrapper: MatrixWrapper
30          """
31          super().__init__(*args, **kwargs)
32
33          self.matrix_wrapper = matrix_wrapper
34          self.setWindowTitle('Define a matrix')
35
36          # === Create the widgets
37
38          self.button_confirm = QtWidgets.QPushButton(self)
39          self.button_confirm.setText('Confirm')
40          self.button_confirm.setEnabled(False)
41          self.button_confirm.clicked.connect(self.confirm_matrix)
42          self.button_confirm.setToolTip('Confirm this as the new matrix<br><b>(Ctrl + Enter)</b>')
43          QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self.button_confirm.click)
44
45          self.button_cancel = QtWidgets.QPushButton(self)
46          self.button_cancel.setText('Cancel')
47          self.button_cancel.clicked.connect(self.close)
48          self.button_cancel.setToolTip('Cancel this definition<br><b>(Ctrl + Q)</b>')
49          QShortcut(QKeySequence('Ctrl+Q'), self).activated.connect(self.button_cancel.click)
50
51          self.label_equals = QtWidgets.QLabel()
52          self.label_equals.setText('=')
53
54          self.letter_combo_box = QtWidgets.QComboBox(self)
55
56          # Everything except I, because that's the identity
57          for letter in ALPHABET_NO_I:
58              self.letter_combo_box.addItem(letter)
59
60          self.letter_combo_box.activated.connect(self.load_matrix)
```

This superclass just has a constructor that subclasses can use. When I added the `DefineAsARotationDialog`

class, I also moved the cancel and confirm buttons into the constructor and added abstract methods that all dialog subclasses must implement.

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

class DefineDialog(QDialog):
...
    def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
...
        # === 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.

```python
# 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)
        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)
```

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

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

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

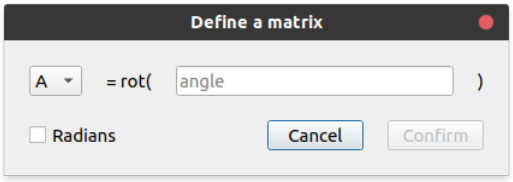


Figure 3.3: The first version of the rotation definition dialog

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

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

17  class LintransMainWindow(QMainWindow):
...
170     def dialog_define_matrix(self, dialog_class: Type[DefineDialog]) -> None:
171         """Open a generic definition dialog to define a new matrix.
172
173         The class for the desired dialog is passed as an argument. We create an
174         instance of this class and the dialog is opened asynchronously and modally
175         (meaning it blocks interaction with the main window) with the proper method
176         connected to the ``dialog.finished`` slot.
177
178         .. note::
179            ``dialog_class`` must subclass :class:`lintrans.gui.dialogs.define_new_matrix.DefineDialog`.
180
181         :param dialog_class: The dialog class to instantiate
182         :type dialog_class: Type[lintrans.gui.dialogs.define_new_matrix.DefineDialog]
183         """
184         # We create a dialog with a deepcopy of the current matrix_wrapper
185         # This avoids the dialog mutating this one
```

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

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

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

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

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

3.3 Visualizing matrices

3.3.1 Asking strangers on the internet for help

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

While looking through the PyQt5 docs, I found a pretty comprehensive explanation of the Qt5 'Graphics View Framework'[28], 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[24].

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

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

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

3.3.2 Creating the plots package

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

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

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

from __future__ import annotations

from PyQt5.QtCore import Qt
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.
```

```
17
18          .. warning:: This class should never be directly instantiated, only subclassed.
19
20          .. note::
21             I would make this class have ``metaclass=abc.ABCMeta``, but I can't because it subclasses ``QWidget``,
22             and a every superclass of a class must have the same metaclass, and ``QWidget`` is not an abstract class.
23          """
24
25          def __init__(self, *args, **kwargs):
26              """Create the widget, passing ``*args`` and ``**kwargs`` to the superclass constructor (``QWidget``)."""
27              super().__init__(*args, **kwargs)
28
29              self.setAutoFillBackground(True)
30
31              # Set the background to white
32              palette = self.palette()
33              palette.setColor(self.backgroundRole(), Qt.white)
34              self.setPalette(palette)
35
36              # Set the gird colour to grey and the axes colour to black
37              self.grid_colour = QColor(128, 128, 128)
38              self.axes_colour = QColor(0, 0, 0)
39
40              self.grid_spacing: int = 50
41              self.line_width: float = 0.4
42
43          @property
44          def w(self) -> int:
45              """Return the width of the widget."""
46              return self.size().width()
47
48          @property
49          def h(self) -> int:
50              """Return the height of the widget."""
51              return self.size().height()
52
53          def paintEvent(self, e: QPaintEvent):
54              """Handle a ``QPaintEvent`` by drawing the widget."""
55              qp = QPainter()
56              qp.begin(self)
57              self.draw_widget(qp)
58              qp.end()
59
60          def draw_widget(self, qp: QPainter):
61              """Draw the grid and axes in the widget."""
62              qp.setRenderHint(QPainter.Antialiasing)
63              qp.setBrush(Qt.NoBrush)
64
65              # Draw the grid
66              qp.setPen(QPen(self.grid_colour, self.line_width))
67
68              # We draw the background grid, centered in the middle
69              # We deliberately exclude the axes - these are drawn separately
70              for x in range(self.w // 2 + self.grid_spacing, self.w, self.grid_spacing):
71                  qp.drawLine(x, 0, x, self.h)
72                  qp.drawLine(self.w - x, 0, self.w - x, self.h)
73
74              for y in range(self.h // 2 + self.grid_spacing, self.h, self.grid_spacing):
75                  qp.drawLine(0, y, self.w, y)
76                  qp.drawLine(0, self.h - y, self.w, self.h - y)
77
78              # Now draw the axes
79              qp.setPen(QPen(self.axes_colour, self.line_width))
80              qp.drawLine(self.w // 2, 0, self.w // 2, self.h)
81              qp.drawLine(0, self.h // 2, self.w, self.h // 2)
82
83
84      class ViewTransformationWidget(TransformationPlotWidget):
85          """This class is used to visualise matrices as transformations."""
86
87          def __init__(self, *args, **kwargs):
88              """Create the widget, passing ``*args`` and ``**kwargs`` to the superclass constructor."""
89              super().__init__(*args, **kwargs)
```

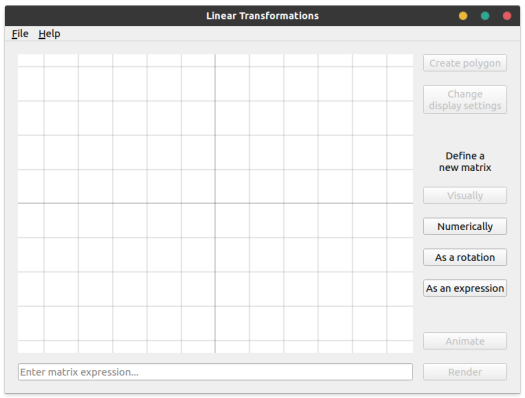


Figure 3.4: The GUI with background axes

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

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

3.3.3 Implementing basis vectors

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

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

class TransformationPlotWidget(QWidget):
...
    @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.

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

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.)
```

```
101
102             self.colour_i = QColor(37, 244, 15)
103             self.colour_j = QColor(8, 8, 216)
104
105             self.width_vector_line = 1
106             self.width_transformed_grid = 0.6
107
108         def transform_by_matrix(self, matrix: MatrixType) -> None:
109             """Transform the plane by the given matrix."""
110             self.point_i = (matrix[0][0], matrix[1][0])
111             self.point_j = (matrix[0][1], matrix[1][1])
112             self.update()
```

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

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

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

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

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

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

Testing this new code shows that it works well.

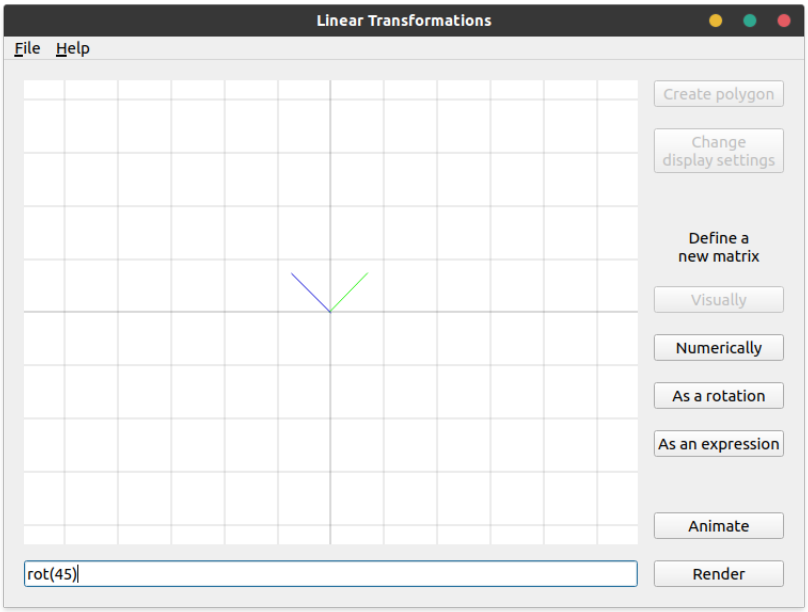


Figure 3.5: Basis vectors drawn for a 45° rotation

3.3.4 Drawing the transformed grid

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

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

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

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

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

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

```
140                     self.height()
141                 )
142                 painter.drawLine(
143                     self.trans_x(-1 * (i + 1) * point_x),
144                     0,
145                     self.trans_x(-1 * (i + 1) * point_x),
146                     self.height()
147                 )
148
149         elif vector_y == 0:
150             painter.drawLine(0, self.trans_y(0), self.width(), self.trans_y(0))
151
152             for i in range(int(max_y / point_y)):
153                 painter.drawLine(
154                     0,
155                     self.trans_y((i + 1) * point_y),
156                     self.width(),
157                     self.trans_y((i + 1) * point_y)
158                 )
159                 painter.drawLine(
160                     0,
161                     self.trans_y(-1 * (i + 1) * point_y),
162                     self.width(),
163                     self.trans_y(-1 * (i + 1) * point_y)
164                 )
```

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

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

96  class ViewTransformationWidget(TransformationPlotWidget):
...
166     def draw_transformed_grid(self, painter: QPainter) -> None:
167         """Draw the transformed version of the grid, given by the unit vectors."""
168         # Draw the unit vectors
169         painter.setPen(QPen(self.colour_i, self.width_vector_line))
170         painter.drawLine(*self.origin, *self.trans_coords(*self.point_i))
171         painter.setPen(QPen(self.colour_j, self.width_vector_line))
172         painter.drawLine(*self.origin, *self.trans_coords(*self.point_j))
173
174         # Draw all the parallel lines
175         painter.setPen(QPen(self.colour_i, self.width_transformed_grid))
176         self.draw_parallel_lines(painter, self.point_i, self.point_j)
177         painter.setPen(QPen(self.colour_j, self.width_transformed_grid))
178         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.

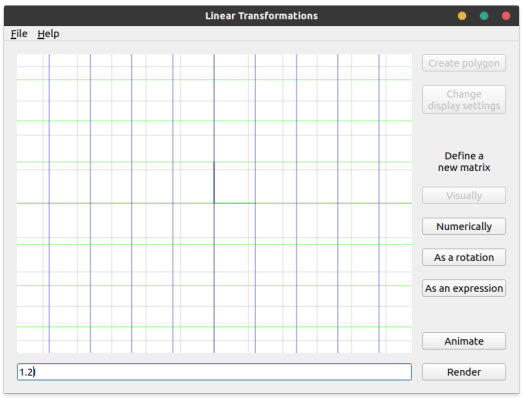


Figure 3.6: Parallel lines being drawn for matrix 1.2**I**

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

```
96  class ViewTransformationWidget(TransformationPlotWidget):
...
126     def draw_parallel_lines(self, painter: QPainter, vector: tuple[float, float], point: tuple[float, float]) ->
        ↪ None:
127         """Draw a set of grid lines parallel to ``vector`` intersecting ``point``."""
128         max_x, max_y = self.grid_corner()
129         vector_x, vector_y = vector
130         point_x, point_y = point
131
132         print(max_x, max_y, vector_x, vector_y, point_x, point_y)
133
134         # We want to use y = mx + c but m = y / x and if either of those are 0, then this
135         # equation is harder to work with, so we deal with these edge cases first
136         if abs(vector_x) < 1e-12 and abs(vector_y) < 1e-12:
137             # If both components of the vector are practically 0, then we can't render any grid lines
138             return
139
140         elif abs(vector_x) < 1e-12:
141             painter.drawLine(self.trans_x(0), 0, self.trans_x(0), self.height())
142
143             for i in range(abs(int(max_x / point_x))):
144                 painter.drawLine(
145                     self.trans_x((i + 1) * point_x),
146                     0,
147                     self.trans_x((i + 1) * point_x),
148                     self.height()
149                 )
150                 painter.drawLine(
151                     self.trans_x(-1 * (i + 1) * point_x),
152                     0,
153                     self.trans_x(-1 * (i + 1) * point_x),
154                     self.height()
155                 )
156
157         elif abs(vector_y) < 1e-12:
158             painter.drawLine(0, self.trans_y(0), self.width(), self.trans_y(0))
159
160             for i in range(abs(int(max_y / point_y))):
161                 painter.drawLine(
162                     0,
163                     self.trans_y((i + 1) * point_y),
164                     self.width(),
165                     self.trans_y((i + 1) * point_y)
166                 )
167                 painter.drawLine(
168                     0,
169                     self.trans_y(-1 * (i + 1) * point_y),
170                     self.width(),
171                     self.trans_y(-1 * (i + 1) * point_y)
172                 )
173
174         else:  # If the line is not horizontal or vertical, then we can use y = mx + c
175             m = vector_y / vector_x
176             c = point_y - m * point_x
177
178             # For c = 0
179             painter.drawLine(
180                 *self.trans_coords(
181                     -1 * max_x,
182                     m * -1 * max_x
183                 ),
184                 *self.trans_coords(
185                     max_x,
186                     m * max_x
187                 )
188             )
189
190             # Count up how many multiples of c we can have without wasting time rendering lines off screen
191             multiples_of_c: int = 0
192             ii: int = 1
193             while True:
194                 y1 = m * max_x + ii * c
```

```
                y2 = -1 * m * max_x + ii * c

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

                else:
                    break

            # 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.

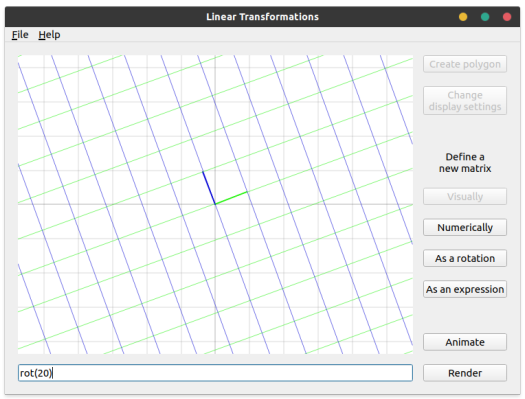


Figure 3.7: An example of a 20° rotation

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.

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

class LintransMainWindow(QMainWindow):
```

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

```
...
238         def animate_expression(self) -> None:
239             """Animate the expression in the input box, and then clear the box."""
240             self.button_render.setEnabled(False)
241             self.button_animate.setEnabled(False)
242 
243             matrix = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())
244             matrix_move = matrix - self.matrix_wrapper['I']
245             steps: int = 100
246 
247             for i in range(0, steps + 1):
248                 self.plot.visualize_matrix_transformation(
249                     self.matrix_wrapper['I'] + (i / steps) * matrix_move
250                 )
251 
252                 self.update()
253                 self.repaint()
254 
255                 time.sleep(0.01)
256 
257             self.button_render.setEnabled(False)
258             self.button_animate.setEnabled(False)
```

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

3.3.6 Preserving determinants

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

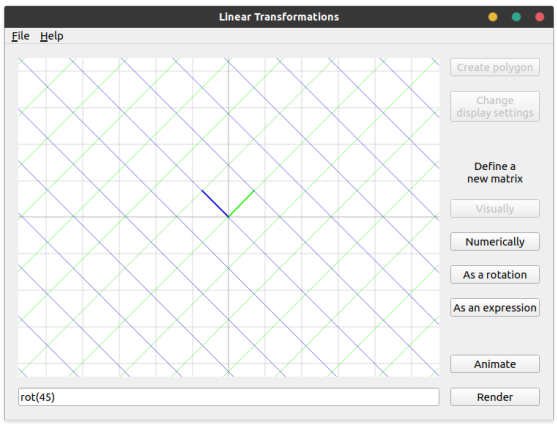


Figure 3.8: What we would expect halfway through a 90° rotation

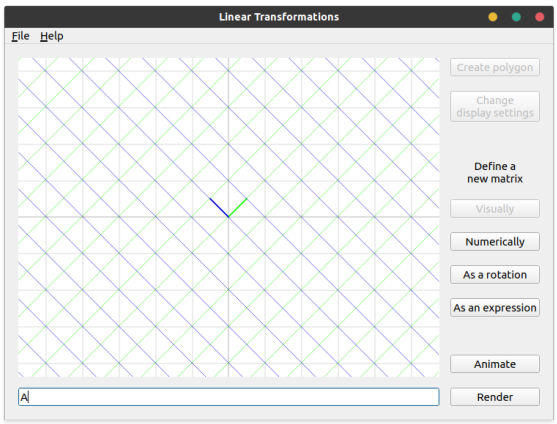


Figure 3.9: What we actually get halfway through a 90° rotation

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

I think I first tried just multiplying $\mathbf{A}$ by $\frac{1}{\det(\mathbf{A})}$ but that didn't work, so I googled it. I found a

post[21] 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 $d = \sqrt{\left|\dfrac{\det(\mathbf{T})}{\det(\mathbf{B})}\right|}$. But $\mathbf{B}$ is defined so that $\det(\mathbf{B}) = 1$, so we can get $d = \sqrt{|\det(\mathbf{T})|}$.

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

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

22  class LintransMainWindow(QMainWindow):
...
240     def animate_expression(self) -> None:
...
245         # Get the target matrix and it's determinant
246         matrix_target = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())
247         det_target = linalg.det(matrix_target)
248
249         identity = self.matrix_wrapper['I']
250         steps: int = 100
251
252         for i in range(0, steps + 1):
253             # This proportion is how far we are through the loop
254             proportion = i / steps
255
256             # matrix_a is the identity plus some part of the target, scaled by the proportion
257             # If we just used matrix_a, then things would animate, but the determinants would be weird
258             matrix_a = identity + proportion * (matrix_target - identity)
259
260             # So to fix the determinant problem, we get the determinant of matrix_a and use it to normalise
261             det_a = linalg.det(matrix_a)
262
263             # For a 2x2 matrix A and a scalar c, we know that det(cA) = c^2 det(A)
264             # We want B = cA such that det(B) = 1, so then we can scale it with the animation
265             # So we get c^2 det(A) = 1 => c = sqrt(1 / abs(det(A)))
266             # Then we scale A down to get a determinant of 1, and call that matrix_b
267             if det_a == 0:
268                 c = 0
269             else:
270                 c = np.sqrt(1 / abs(det_a))
271
272             matrix_b = c * matrix_a
273
274             # matrix_c is the final matrix that we transform by
275             # It's B, but we scale it up over time to have the target determinant
276
277             # We want some C = dB such that det(C) is some target determinant T
278             # det(dB) = d^2 det(B) = T => d = sqrt(abs(T / det(B))
279             # But we defined B to have det 1, so we can ignore it there
280
281             # We're also subtracting 1 and multiplying by the proportion and then adding one
282             # This just scales the determinant along with the animation
283             scalar = 1 + proportion * (np.sqrt(abs(det_target)) - 1)
284
285             matrix_c = scalar * matrix_b
286
287             self.plot.visualize_matrix_transformation(matrix_c)
288
289             self.repaint()
290             time.sleep(0.01)
```

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

3.4 Improving the GUI

3.4.1 Fixing rendering

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

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

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

118 class VectorGridPlot(BackgroundPlot):
...
150     def draw_parallel_lines(self, painter: QPainter, vector: tuple[float, float], point: tuple[float, float]) ->
        ↪ None:
...
203         else:  # If the line is not horizontal or vertical, then we can use y = mx + c
204             m = vector_y / vector_x
205             c = point_y - m * point_x
206 
207             # For c = 0
208             painter.drawLine(
209                 *self.trans_coords(
210                     -1 * max_x,
211                     m * -1 * max_x
212                 ),
213                 *self.trans_coords(
214                     max_x,
215                     m * max_x
216                 )
217             )
218 
219             # We keep looping and increasing the multiple of c until we stop drawing lines on the canvas
220             multiple_of_c = 1
221             while self.draw_pair_of_oblique_lines(painter, m, multiple_of_c * c):
222                 multiple_of_c += 1
```

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

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

118 class VectorGridPlot(BackgroundPlot):
...
224     def draw_pair_of_oblique_lines(self, painter: QPainter, m: float, c: float) -> bool:
225         """Draw a pair of oblique lines, using the equation y = mx + c.
226 
227         This method just calls :meth:`draw_oblique_line` with ``c`` and ``-c``,
228         and returns True if either call returned True.
229 
230         :param QPainter painter: The ``QPainter`` object to use for drawing the vectors and grid lines
231         :param float m: The gradient of the lines to draw
232         :param float c: The y-intercept of the lines to draw. We use the positive and negative versions
233         :returns bool: Whether we were able to draw any lines on the canvas
234         """
235         return any([
236             self.draw_oblique_line(painter, m, c),
237             self.draw_oblique_line(painter, m, -c)
238         ])
239 
```

```
240         def draw_oblique_line(self, painter: QPainter, m: float, c: float) -> bool:
241             """Draw an oblique line, using the equation y = mx + c.
242 
243             We only draw the part of the line that fits within the canvas, returning True if
244             we were able to draw a line within the boundaries, and False if we couldn't draw a line
245 
246             :param QPainter painter: The ``QPainter`` object to use for drawing the vectors and grid lines
247             :param float m: The gradient of the line to draw
248             :param float c: The y-intercept of the line to draw
249             :returns bool: Whether we were able to draw a line on the canvas
250             """
251             max_x, max_y = self.grid_corner()
252 
253             # These variable names are shortened for convenience
254             # myi is max_y_intersection, mmyi is minus_max_y_intersection, etc.
255             myi = (max_y - c) / m
256             mmyi = (-max_y - c) / m
257             mxi = max_x * m + c
258             mmxi = -max_x * m + c
259 
260             # The inner list here is a list of coords, or None
261             # If an intersection fits within the bounds, then we keep its coord,
262             # else it is None, and then gets discarded from the points list
263             # By the end, points is a list of two coords, or an empty list
264             points: list[tuple[float, float]] = [
265                 x for x in [
266                     (myi, max_y) if -max_x < myi < max_x else None,
267                     (mmyi, -max_y) if -max_x < mmyi < max_x else None,
268                     (max_x, mxi) if -max_y < mxi < max_y else None,
269                     (-max_x, mmxi) if -max_y < mmxi < max_y else None
270                 ] if x is not None
271             ]
272 
273             # If no intersections fit on the canvas
274             if len(points) < 2:
275                 return False
276 
277             # If we can, then draw the line
278             else:
279                 painter.drawLine(
280                     *self.trans_coords(*points[0]),
281                     *self.trans_coords(*points[1])
282                 )
283                 return True
```

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

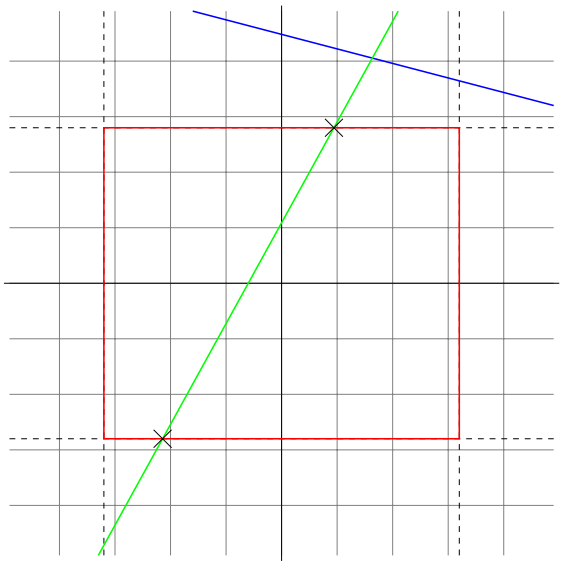


Figure 3.10: Two example lines and the viewport box

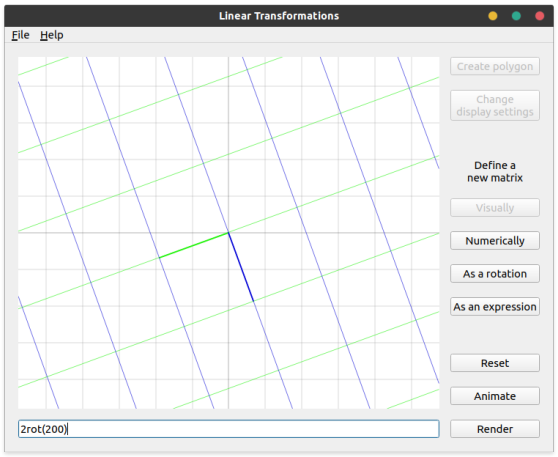


Figure 3.11: A demonstration of the new oblique lines system.

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

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

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

3.4.2 Adding vector arrowheads

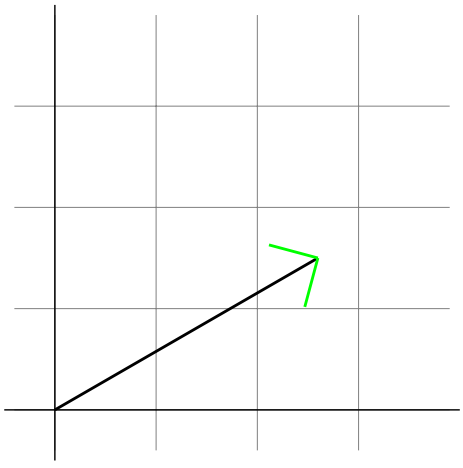


Figure 3.12: An example of a vector with the arrowheads highlighted in green

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

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

So I started googling and found a very nice algorithm on `csharphelper.com`[43], which I adapted for Python.

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

class VisualizeTransformationWidget(VectorGridPlot):
...
    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))

    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()`.

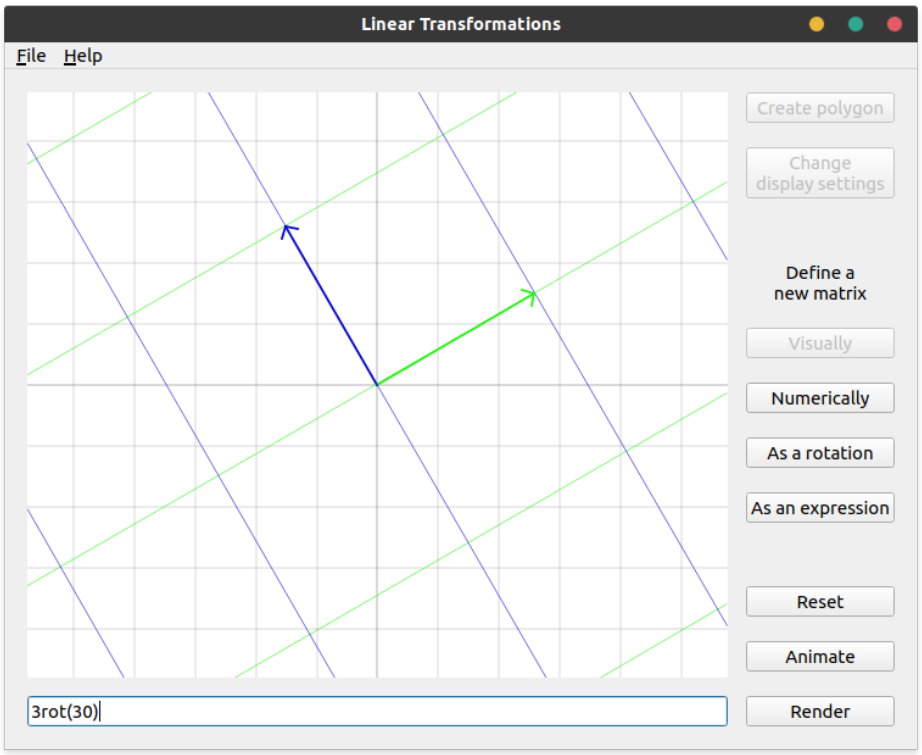


Figure 3.13: An example of the i and j vectors with arrowheads

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

12  class BackgroundPlot(QWidget):
...
27      default_grid_spacing: int = 50
28
29      def __init__(self, *args, **kwargs):
30          """Create the widget and setup backend stuff for rendering.
31
32          .. note:: ``*args`` and ``**kwargs`` are passed the superclass constructor (``QWidget``).
33          """
34          super().__init__(*args, **kwargs)
35
36          self.setAutoFillBackground(True)
37
38          # Set the background to white
39          palette = self.palette()
40          palette.setColor(self.backgroundRole(), Qt.white)
41          self.setPalette(palette)
42
43          # Set the gird colour to grey and the axes colour to black
44          self.colour_background_grid = QColor(128, 128, 128)
45          self.colour_background_axes = QColor(0, 0, 0)
46
47          self.grid_spacing = BackgroundPlot.default_grid_spacing
```

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

To actually allow for zooming, I had to implement the `wheelEvent()` method in `BackgroundPlot` to listen for mouse wheel events. After reading through the docs for the `QWheelEvent` class[33], I learned how to handle this event.

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

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

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

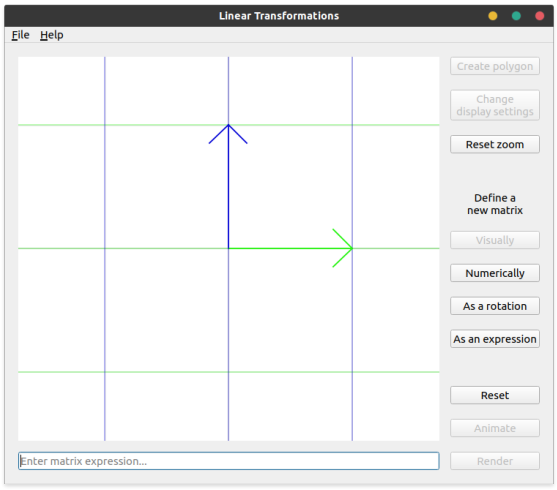


Figure 3.14: The GUI zoomed in a bit

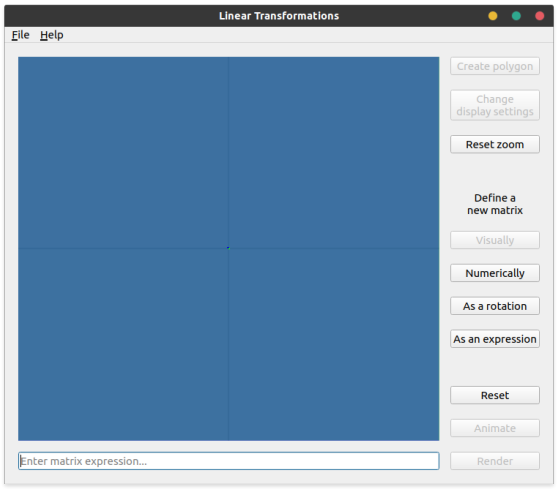


Figure 3.15: The GUI zoomed out as far as possible

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

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

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

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

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

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

```
12  class VisualizeTransformationWidget(VectorGridPlot):
...
54      def draw_arrowhead_away_from_origin(self, painter: QPainter, point: tuple[float, float]) -> None:
...
68          vector_length = np.sqrt(x * x + y * y)
69          nx = x / vector_length
70          ny = y / vector_length
71
72          # We choose a length and find the steps in the x and y directions
73          length = min(
74              self.arrowhead_length * self.default_grid_spacing / self.grid_spacing,
75              vector_length
76          )
```

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

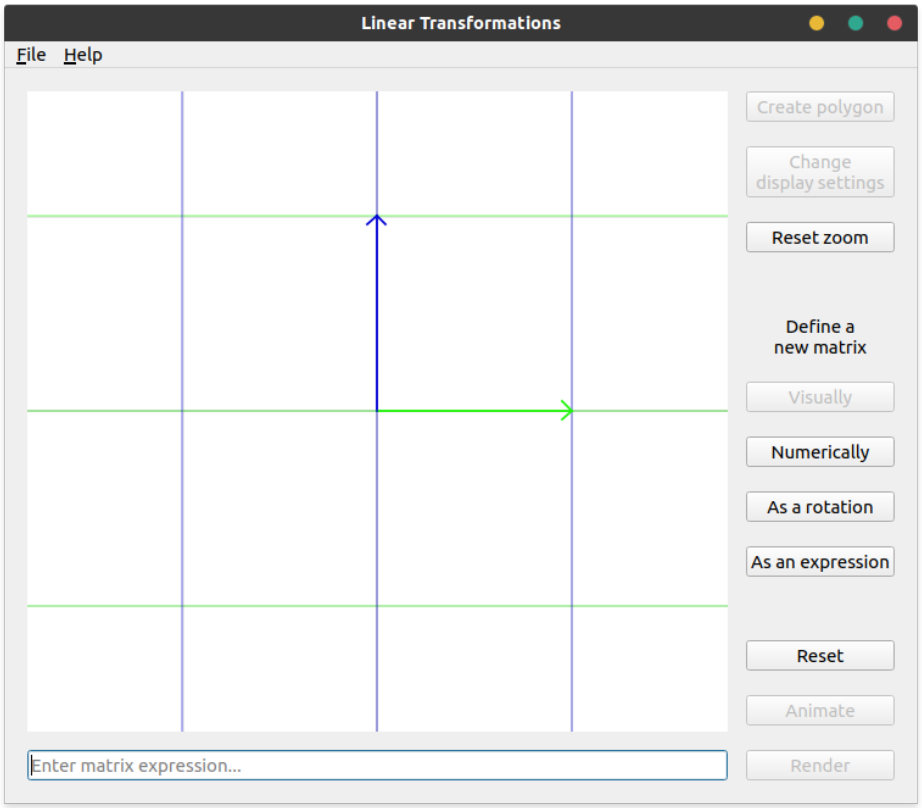


Figure 3.16: The arrowheads adjusted for zoom level

3.4.4 Animation blocks zooming

The biggest problem with this new zoom feature is that when animating between matrices, the user is unable to zoom. This is because when `LintransMainWindow.animate_expression()` is called, it uses Python's standard library `time.sleep()` function to delay each frame, which prevents Qt from handling user interaction while we're animating. This was a problem.

I did some googling and found a helpful post on StackOverflow[17] 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[44], 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[36, 38] and decided that it would be better to just queue a repaint by calling `update()` on the plot rather than immediately repaint with `repaint()`, and then call `QApplication.processEvents()` to process the pending events on the main thread. This is a nicer way of repainting, which reduces potential flickering issues, and using `QThread.msleep()` allows for asynchronous processing and therefore non-blocking animation.

3.4.5 Rank 1 transformations

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

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

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

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

132 class VectorGridPlot(BackgroundPlot):
...
164     def draw_parallel_lines(self, painter: QPainter, vector: tuple[float, float], point: tuple[float, float]) ->
        ↪ None:
...
177         # If the determinant is 0
178         if abs(vector_x * point_y - vector_y * point_x) < 1e-12:
179             rank = np.linalg.matrix_rank(
180                 np.array([
181                     [vector_x, point_x],
182                     [vector_y, point_y]
183                 ])
184             )
185
186             # If the matrix is rank 1, then we can draw the column space line
187             if rank == 1:
188                 self.draw_oblique_line(painter, vector_y / vector_x, 0)
189
190             # If the rank is 0, then we don't draw any lines
191             else:
192                 return
```

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

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

22 class LintransMainWindow(QMainWindow):
...
262     def animate_expression(self) -> None:
...
```

```
274             for i in range(0, steps + 1):
...
307                 # If we're animating towards a det 0 matrix, then we don't want to scale the
308                 # determinant with the animation, because this makes the process not work
309                 # I'm doing this here rather than wrapping the whole animation logic in an
310                 # if block mainly because this looks nicer than an extra level of indentation
311                 # The extra processing cost is negligible thanks to NumPy's optimizations
312                 if det_target == 0:
313                     matrix_c = matrix_a
314                 else:
315                     matrix_c = scalar * matrix_b
```

3.4.6 Matrices that are too big

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

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

I'd previously created a `LintransMainWindow.show_error_message()` method for telling the user when they try to take the inverse of a singular matrix[11].

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

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

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

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

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

```
24  class LintransMainWindow(QMainWindow):
...
407     def is_matrix_too_big(self, matrix: MatrixType) -> bool:
408         """Check if the given matrix will actually fit onto the canvas.
409
410         Convert the elements of the matrix to canvas coords and make sure they fit within Qt's 32-bit integer limit.
411
412         :param MatrixType matrix: The matrix to check
413         :returns bool: Whether the matrix fits on the canvas
414         """
415         coords: list[tuple[int, int]] = [self.plot.trans_coords(*vector) for vector in matrix.T]
416
417         for x, y in coords:
418             if not (-2147483648 <= x <= 2147483647 and -2147483648 <= y <= 2147483647):
419                 return True
420
421         return False
```

3.4.7 Creating the `DefineVisuallyDialog`

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

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

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

```
178
179         def show_matrix(self, index: int) -> None:
180             """Show the selected matrix on the plot. If the matrix is None, show the identity."""
181             matrix = self.matrix_wrapper[ALPHABET_NO_I[index]]
182
183             if matrix is None:
184                 matrix = self.matrix_wrapper['I']
185
186             self.plot.visualize_matrix_transformation(matrix)
187             self.plot.update()
188
189         def confirm_matrix(self) -> None:
```

This `DefineVisuallyDialog` class just implements the normal methods needed for a `DefineDialog` and has a `plot` attribute to handle drawing graphics and handling mouse movement. After creating the `DefineVisuallyWidget` as a skeleton and doing some more research in the Qt5 docs[35], I renamed the `trans_coords()` methods to `canvas_coords()` to make the intent more clear, and created a `grid_coords()` method.

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

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

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

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

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

```
77              button = event.button()
78
79              if button != Qt.LeftButton:
80                  event.ignore()
81                  return
82
83              for point in (self.point_i, self.point_j):
84                  px, py = self.canvas_coords(*point)
85                  if abs(px - mx) <= self.epsilon and abs(py - my) <= self.epsilon:
86                      self.dragged_point = point[0], point[1]
87
88              event.accept()
89
90          def mouseReleaseEvent(self, event: QMouseEvent) -> None:
91              """Handle a QMouseEvent when the user release a button."""
92              if event.button() == Qt.LeftButton:
93                  self.dragged_point = None
94                  event.accept()
95              else:
96                  event.ignore()
97
98          def mouseMoveEvent(self, event: QMouseEvent) -> None:
99              """Handle the mouse moving on the canvas."""
100             mx = event.x()
101             my = event.y()
102
103             if self.dragged_point is not None:
104                 x, y = self.grid_coords(mx, my)
105
106                 if self.dragged_point == self.point_i:
107                     self.point_i = x, y
108
109                 elif self.dragged_point == self.point_j:
110                     self.point_j = x, y
111
112                 self.dragged_point = x, y
113
114                 self.update()
115
116                 print(self.dragged_point)
117                 print(self.point_i, self.point_j)
118
119                 event.accept()
120
121             event.ignore()
```

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

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

3.4.8 Fixing a division by zero bug

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

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

class VectorGridPlot(BackgroundPlot):
...
    def draw_parallel_lines(self, painter: QPainter, vector: tuple[float, float], point: tuple[float, float]) ->
    ↪ None:
...
        if abs(vector_x * point_y - vector_y * point_x) < 1e-12:
...
            # 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 $\mathbf{I}$ to the target matrix $\mathbf{T}$. This means it resets the plot at the start. I eventually want an applicative animation system, where the matrix in the box is applied to the current scene. But I also want an option for a transitional animation, where the program animates from the start matrix $\mathbf{S}$ to the target matrix $\mathbf{T}$, and this seems easier to implement, so I'll do it first.

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

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

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

class LintransMainWindow(QMainWindow):
...
    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:
```

```
282                 matrix_target = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())
283
284             except linalg.LinAlgError:
285                 self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
286                 return
287
288             matrix_start: MatrixType = np.array([
289                 [self.plot.point_i[0], self.plot.point_j[0]],
290                 [self.plot.point_i[1], self.plot.point_j[1]]
291             ])
292
293             self.animate_between_matrices(matrix_start, matrix_target)
294
295             self.button_render.setEnabled(True)
296             self.button_animate.setEnabled(True)
297
298         def animate_between_matrices(self, matrix_start: MatrixType, matrix_target: MatrixType, steps: int = 100) ->
    ↪ None:
299             """Animate from the start matrix to the target matrix."""
300             det_target = linalg.det(matrix_target)
301             det_start = linalg.det(matrix_start)
302
303             for i in range(0, steps + 1):
304                 # This proportion is how far we are through the loop
305                 proportion = i / steps
306
307                 # matrix_a is the start matrix plus some part of the target, scaled by the proportion
308                 # If we just used matrix_a, then things would animate, but the determinants would be weird
309                 matrix_a = matrix_start + proportion * (matrix_target - matrix_start)
310
311                 # So to fix the determinant problem, we get the determinant of matrix_a and use it to normalise
312                 det_a = linalg.det(matrix_a)
313
314                 # For a 2x2 matrix A and a scalar c, we know that det(cA) = c^2 det(A)
315                 # We want B = cA such that det(B) = det(S), where S is the start matrix,
316                 # so then we can scale it with the animation, so we get
317                 # det(cA) = c^2 det(A) = det(S) => c = sqrt(abs(det(S) / det(A)))
318                 # Then we scale A to get the determinant we want, and call that matrix_b
319                 if det_a == 0:
320                     c = 0
321                 else:
322                     c = np.sqrt(abs(det_start / det_a))
323
324                 matrix_b = c * matrix_a
325                 det_b = linalg.det(matrix_b)
326
327                 # matrix_c is the final matrix that we then render for this frame
328                 # It's B, but we scale it over time to have the target determinant
329
330                 # We want some C = dB such that det(C) is some target determinant T
331                 # det(dB) = d^2 det(B) = T => d = sqrt(abs(T / det(B))
332
333                 # We're also subtracting 1 and multiplying by the proportion and then adding one
334                 # This just scales the determinant along with the animation
335                 scalar = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
336
337                 # If we're animating towards a det 0 matrix, then we don't want to scale the
338                 # determinant with the animation, because this makes the process not work
339                 # I'm doing this here rather than wrapping the whole animation logic in an
340                 # if block mainly because this looks nicer than an extra level of indentation
341                 # The extra processing cost is negligible thanks to NumPy's optimizations
342                 if det_target == 0:
343                     matrix_c = matrix_a
344                 else:
345                     matrix_c = scalar * matrix_b
346
347                 if self.is_matrix_too_big(matrix_c):
348                     self.show_error_message('Matrix too big', "This matrix doesn't fit on the canvas")
349                     return
350
351                 self.plot.visualize_matrix_transformation(matrix_c)
352
353                 # We schedule the plot to be updated, tell the event loop to
```

```
354                 # process events, and asynchronously sleep for 10ms
355                 # This allows for other events to be processed while animating, like zooming in and out
356                 self.plot.update()
```

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

3.4.10 Allowing for sequential animation with commas

Applicative animation has two main forms. There's the version where a standard matrix expression gets applied to the current scene, and the kind where the user defines a sequence of matrices and we animate through the sequence, applying one at a time. Both of these are referenced in success criterion 5.

I want the user to be able to decide if they want applicative animation or transitional animation, so I'll need to create some form of display settings. However, transitional animation doesn't make much sense for sequential animation[12], so I can implement this now.

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

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

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

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

25  class LintransMainWindow(QMainWindow):
...
284     def animate_expression(self) -> None:
285         """Animate from the current matrix to the matrix in the expression box."""
286         self.button_render.setEnabled(False)
287         self.button_animate.setEnabled(False)
288
289         matrix_start: MatrixType = np.array([
290             [self.plot.point_i[0], self.plot.point_j[0]],
291             [self.plot.point_i[1], self.plot.point_j[1]]
292         ])
293
294         text = self.lineedit_expression_box.text()
295
296         # If there's commas in the expression, then we want to animate each part at a time
297         if ',' in text:
298             current_matrix = matrix_start
299
300             # For each expression in the list, right multiply it by the current matrix,
301             # and animate from the current matrix to that new matrix
302             for expr in text.split(',')[::-1]:
303                 new_matrix = self.matrix_wrapper.evaluate_expression(expr) @ current_matrix
304
305                 self.animate_between_matrices(current_matrix, new_matrix)
306                 current_matrix = new_matrix
307
308                 # Here we just redraw and allow for other events to be handled while we pause
309                 self.plot.update()
310                 QApplication.processEvents()
311                 QThread.msleep(500)
```

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

```
312
313             # If there's no commas, then just animate directly from the start to the target
314             else:
315                 # Get the target matrix and it's determinant
316                 try:
317                     matrix_target = self.matrix_wrapper.evaluate_expression(text)
318
319                 except linalg.LinAlgError:
320                     self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
321                     return
322
323                 self.animate_between_matrices(matrix_start, matrix_target)
324
325             self.update_render_buttons()
```

We're deliberately not checking if the sub-expressions are valid here. We would normally validate the expression in `LintransMainWindow.update_render_buttons()` and only allow the user to render or animate an expression if it's valid. Now we have to check all the sub-expressions if the expression contains commas. Additionally, we can only animate these expressions with commas in them, so rendering should be disabled when the expression contains commas.

Compare the old code to the new code:

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

25  class LintransMainWindow(QMainWindow):
...
243     def update_render_buttons(self) -> None:
244         """Enable or disable the render and animate buttons according to whether the matrix expression is valid."""
245         valid = self.matrix_wrapper.is_valid_expression(self.lineedit_expression_box.text())
246         self.button_render.setEnabled(valid)
247         self.button_animate.setEnabled(valid)


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

25  class LintransMainWindow(QMainWindow):
...
243     def update_render_buttons(self) -> None:
244         """Enable or disable the render and animate buttons according to whether the matrix expression is valid."""
245         text = self.lineedit_expression_box.text()
246
247         if ',' in text:
248             self.button_render.setEnabled(False)
249
250             valid = all(self.matrix_wrapper.is_valid_expression(x) for x in text.split(','))
251             self.button_animate.setEnabled(valid)
252
253         else:
254             valid = self.matrix_wrapper.is_valid_expression(text)
255             self.button_render.setEnabled(valid)
256             self.button_animate.setEnabled(valid)
```

3.5 Adding display settings

3.5.1 Creating the dataclass (and implementing applicative animation)

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

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

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

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

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

from dataclasses import dataclass


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

    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

class LintransMainWindow(QMainWindow):
...
    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()

```

```
298             # If there's commas in the expression, then we want to animate each part at a time
299             if ',' in text:
300                 current_matrix = matrix_start
301 
302                 # For each expression in the list, right multiply it by the current matrix,
303                 # and animate from the current matrix to that new matrix
304                 for expr in text.split(',')[::-1]:
305                     new_matrix = self.matrix_wrapper.evaluate_expression(expr) @ current_matrix
306 
307                     self.animate_between_matrices(current_matrix, new_matrix)
308                     current_matrix = new_matrix
309 
310                     # Here we just redraw and allow for other events to be handled while we pause
311                     self.plot.update()
312                     QApplication.processEvents()
313                     QThread.msleep(self.display_settings.animation_pause_length)
314 
315             # If there's no commas, then just animate directly from the start to the target
316             else:
317                 # Get the target matrix and it's determinant
318                 try:
319                     matrix_target = self.matrix_wrapper.evaluate_expression(text)
320 
321                 except linalg.LinAlgError:
322                     self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
323                     return
324 
325                 # The concept of applicative animation is explained in /gui/settings.py
326                 if self.display_settings.applicative_animation:
327                     matrix_target = matrix_target @ matrix_start
328 
329                 self.animate_between_matrices(matrix_start, matrix_target)
330 
331             self.update_render_buttons()
```

Lines 327 are very important here. I included applicative animation as an option in the display settings because once I'd implemented animating from one matrix to another, it was very easy to implement applicative animation.

The user will input whatever matrix they wanted to apply to the current scene. Let's call that target matrix $\mathbf{T}$. The matrix representing the starting state of the viewport is $\mathbf{S}$. Animating from $\mathbf{S}$ to $\mathbf{T}$ is a transitional animation, but an applicative animation is simply animating from $\mathbf{S}$ to $\mathbf{TS}$, so we can just say `matrix_target = matrix_target @ matrix_start` on line 327 (where `@` is the matrix multiplication operator), and continue as normal.

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.

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

26  class LintransMainWindow(QMainWindow):
...
333     def animate_between_matrices(self, matrix_start: MatrixType, matrix_target: MatrixType, steps: int = 100) ->
        ↪ None:
334         """Animate from the start matrix to the target matrix."""
335         det_target = linalg.det(matrix_target)
336         det_start = linalg.det(matrix_start)
337 
338         for i in range(0, steps + 1):
339             # This proportion is how far we are through the loop
340             proportion = i / steps
341 
342             # matrix_a is the start matrix plus some part of the target, scaled by the proportion
343             # If we just used matrix_a, then things would animate, but the determinants would be weird
344             matrix_a = matrix_start + proportion * (matrix_target - matrix_start)
345 
346             if self.display_settings.animate_determinant and det_target != 0:
347                 # To fix the determinant problem, we get the determinant of matrix_a and use it to normalise
```

```
348                 det_a = linalg.det(matrix_a)
349
350                 # For a 2x2 matrix A and a scalar c, we know that det(cA) = c^2 det(A)
351                 # We want B = cA such that det(B) = det(S), where S is the start matrix,
352                 # so then we can scale it with the animation, so we get
353                 # det(cA) = c^2 det(A) = det(S) => c = sqrt(abs(det(S) / det(A)))
354                 # Then we scale A to get the determinant we want, and call that matrix_b
355                 if det_a == 0:
356                     c = 0
357                 else:
358                     c = np.sqrt(abs(det_start / det_a))
359
360                 matrix_b = c * matrix_a
361                 det_b = linalg.det(matrix_b)
362
363                 # matrix_to_render is the final matrix that we then render for this frame
364                 # It's B, but we scale it over time to have the target determinant
365
366                 # We want some C = dB such that det(C) is some target determinant T
367                 # det(dB) = d^2 det(B) = T => d = sqrt(abs(T / det(B))
368
369                 # We're also subtracting 1 and multiplying by the proportion and then adding one
370                 # This just scales the determinant along with the animation
371                 scalar = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
372                 matrix_to_render = scalar * matrix_b
373
374             else:
375                 matrix_to_render = matrix_a
376
377             if self.is_matrix_too_big(matrix_to_render):
378                 self.show_error_message('Matrix too big', "This matrix doesn't fit on the canvas")
379                 return
380
381             self.plot.visualize_matrix_transformation(matrix_to_render)
382
383             # We schedule the plot to be updated, tell the event loop to
384             # process events, and asynchronously sleep for 10ms
385             # This allows for other events to be processed while animating, like zooming in and out
386             self.plot.update()
387             QApplication.processEvents()
388             QThread.msleep(1000 // steps)
```

3.5.2 Creating the settings dialog

Display settings are good, but useless on their own. My next step was to add a settings dialog that would allow the user to edit these settings.

I first had to create the dialog class itself, so I created the `SettingsDialog` superclass first, so that I could use it for global settings in the future, as well as the specific `DisplaySettingsDialog` subclass now.

As far as I know, a dialog in Qt can't really return a value when it's closed[13], so the dialog keeps a public instance attribute for the `DisplaySettings` class itself, and then the main window can copy that instance attribute when the dialog is closed.

```
# b1ba4adc3c7723c95b490e831e651a7781af7d99
# src/lintrans/gui/dialogs/settings.py

1   """This module provides dialogs to edit settings within the app."""
2
3   from __future__ import annotations
4
5   import abc
```

[13]This is because Qt uses a system of event loops, so the main window continues executing its main loop while the dialog is doing the same. That means that the main window can't wait around for the dialog to close, so nothing can be returned from it.

```
import copy

from PyQt5 import QtWidgets
from PyQt5.QtCore import Qt
from PyQt5.QtGui import QIntValidator, QKeySequence
from PyQt5.QtWidgets import QCheckBox, QDialog, QHBoxLayout, QShortcut, QSizePolicy, QSpacerItem, QVBoxLayout

from lintrans.gui.settings import DisplaySettings


class SettingsDialog(QDialog):
    """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, display_settings: DisplaySettings, *args, **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')

        # === Create the widgets
```

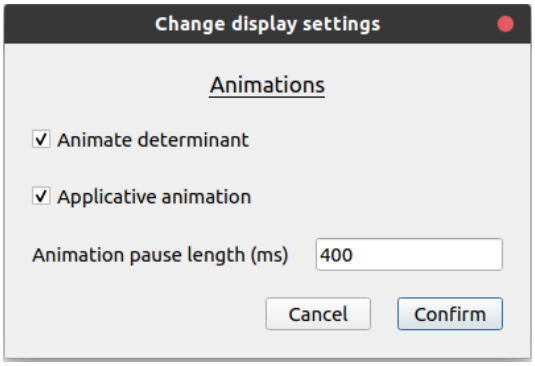
```
79
80              font_label = self.font()
81              font_label.setUnderline(True)
82              font_label.setPointSize(int(font_label.pointSize() * 1.2))
83
84              self.label_animations = QtWidgets.QLabel(self)
85              self.label_animations.setText('Animations')
86              self.label_animations.setAlignment(Qt.AlignCenter)
87              self.label_animations.setFont(font_label)
88
89              self.checkbox_animate_determinant = QCheckBox(self)
90              self.checkbox_animate_determinant.setText('Animate determinant')
91              self.checkbox_animate_determinant.setToolTip('Smoothly animate the determinant during animation')
92
93              self.checkbox_applicative_animation = QCheckBox(self)
94              self.checkbox_applicative_animation.setText('Applicative animation')
95              self.checkbox_applicative_animation.setToolTip(
96                  'Animate the new transformation applied to the current one,\n'
97                  'rather than just that transformation on its own'
98              )
99
100             self.label_animation_pause_length = QtWidgets.QLabel(self)
101             self.label_animation_pause_length.setText('Animation pause length (ms)')
102             self.label_animation_pause_length.setToolTip(
103                 'How many milliseconds to pause for in comma-separated animations'
104             )
105
106             self.lineedit_animation_pause_length = QtWidgets.QLineEdit(self)
107             self.lineedit_animation_pause_length.setValidator(QIntValidator(1, 999, self))
108
109             # === Arrange the widgets
110
111             self.hlay_animation_pause_length = QHBoxLayout()
112             self.hlay_animation_pause_length.addWidget(self.label_animation_pause_length)
113             self.hlay_animation_pause_length.addWidget(self.lineedit_animation_pause_length)
114
115             self.vlay_options.addWidget(self.label_animations)
116             self.vlay_options.addWidget(self.checkbox_animate_determinant)
117             self.vlay_options.addWidget(self.checkbox_applicative_animation)
118             self.vlay_options.addLayout(self.hlay_animation_pause_length)
119
120             # Finally, we load the current settings
121             self.load_settings()
122
123         def load_settings(self) -> None:
124             """Load the current display settings into the widgets."""
125             self.checkbox_animate_determinant.setChecked(self.display_settings.animate_determinant)
126             self.checkbox_applicative_animation.setChecked(self.display_settings.applicative_animation)
127             self.lineedit_animation_pause_length.setText(str(self.display_settings.animation_pause_length))
128
129         def confirm_settings(self) -> None:
130             """Build a :class:`lintrans.gui.settings.DisplaySettings` object and assign it."""
131             self.display_settings.animate_determinant = self.checkbox_animate_determinant.isChecked()
132             self.display_settings.applicative_animation = self.checkbox_applicative_animation.isChecked()
133             self.display_settings.animation_pause_length = int(self.lineedit_animation_pause_length.text())
134
135             self.accept()
```

I then just had to enable the button in the main GUI and implement the method to open the new dialog. I have to use a lambda to capture the local `dialog` variable, but a separate method to actually assign its display settings, since Python doesn't allow assignments in lambda expressions.

```
# b1ba4adc3c7723c95b490e831e651a7781af7d99
# src/lintrans/gui/main_window.py

27  class LintransMainWindow(QMainWindow):
...
436         def dialog_change_display_settings(self) -> None:
437             """Open the dialog to change the display settings."""
438             dialog = DisplaySettingsDialog(self.display_settings, self)
```

```
439             dialog.open()
440             dialog.finished.connect(lambda: self._assign_display_settings(dialog.display_settings))
441
442         def _assign_display_settings(self, display_settings: DisplaySettings) -> None:
443             """Assign a new value to ``self.display_settings``."""
444             self.display_settings = display_settings
```



The `dialog.finished` signal on line 440 should really be `dialog.accepted`. Currently, we re-assign the display settings whenever the dialog is closed in any way. Really, we should only re-assign them when the user hits the confirm button, but trying to cancel the changes will currently save them. This was a silly mistake and I fixed it along with some similar signal-related bugs a few weeks later. See §3.9.1.

Figure 3.17: The display settings dialog

3.5.3 Fixing a bug with transitional animation

While playing around with these new display settings, I encountered a bug with transitional animation. When you animate an expression with transitional animation and then animate the same thing again, nothing happens. This is because the app tries to transition from the starting position to the target position, but they are the same position, so nothing moves.

To fix this, I had to check if the start and target matrices were the same (within floating point error), and then reset the viewport to the identity first, before animating to the target as requested.

```
# fa4a65540749e84b750ddea8abfd36a86c224b47
# src/lintrans/gui/main_window.py

27  class LintransMainWindow(QMainWindow):
...
285     def animate_expression(self) -> None:
...
315         else:
...
328             # If we want a transitional animation and we're animating the same matrix, then restart the animation
329             # We use this check rather than equality because of small floating point errors
330             elif (matrix_start - matrix_target < 1e-12).all():
331                 matrix_start = self.matrix_wrapper['I']
332
333                 # We pause here for 200 ms to make the animation look a bit nicer
334                 self.plot.visualize_matrix_transformation(matrix_start)
335                 self.plot.update()
336                 QApplication.processEvents()
337                 QThread.msleep(200)
```

I later found a bug on line 330. If we subtract the start and target matrices and get a matrix of all negative numbers (rather than all zeroes, which is what I wanted to check for), then the if condition will still be true. That means that some completely different matrices can be considered the same, and the viewport will reset before animating them. To fix this, I can simply take the absolute value.

```
# 3c490c48a0f4017ab8ee9cf471a65c251817b00e
# src/lintrans/gui/main_window.py

333             elif (abs(matrix_start - matrix_target) < 1e-12).all():
```

3.5.4 Adding the determinant parallelogram

The determinant can be represented as the area of the parallelogram formed by the basis vectors. This would be good to visualize in the app.

To do that, I had to add a setting to the display settings, create a function to actually draw it in `VectorGridPlot`, and call that function from `paintEvent()`.

```
# e9e76c1d4f28452efc6ae18afb936616006fd04a
# src/lintrans/gui/settings.py

class DisplaySettings:
...

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

```
# e9e76c1d4f28452efc6ae18afb936616006fd04a
# src/lintrans/gui/plots/classes.py

class VectorGridPlot(BackgroundPlot):
...

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

        brush = QBrush(QColor(16, 235, 253, alpha=128), Qt.SolidPattern)
        painter.fillPath(path, brush)
```

```
# e9e76c1d4f28452efc6ae18afb936616006fd04a
# src/lintrans/gui/plots/widgets.py

class VisualizeTransformationWidget(VectorGridPlot):
...

    def paintEvent(self, event: QPaintEvent) -> None:
        """Handle a ``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.draw_transformed_grid(painter)
        self.draw_vector_arrowheads(painter)

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

        painter.end()
        event.accept()
```

I then wanted to change the determinant parallelogram to be blue when it's positive and red when it's negative. I did this by just checking the sign of the determinant and changing the colour accordingly.

```
# cc75c7dc85e941540f7e98fe027d0657ad5462b8
# src/lintrans/gui/plots/classes.py

class VectorGridPlot(BackgroundPlot):
```

```
...
385         def draw_determinant_parallelogram(self, painter: QPainter) -> None:
386             """Draw the parallelogram of the determinant of the matrix."""
387             det = np.linalg.det(np.array([
388                 [self.point_i[0], self.point_j[0]],
389                 [self.point_i[1], self.point_j[1]]
390             ]))
391
392             if det == 0:
393                 return
394
395             path = QPainterPath()
396             path.moveTo(*self.canvas_origin)
397             path.lineTo(*self.canvas_coords(*self.point_i))
398             path.lineTo(*self.canvas_coords(self.point_i[0] + self.point_j[0], self.point_i[1] + self.point_j[1]))
399             path.lineTo(*self.canvas_coords(*self.point_j))
400
401             color = (16, 235, 253) if det > 0 else (253, 34, 16)
402             brush = QBrush(QColor(*color, alpha=128), Qt.SolidPattern)
403
404             painter.fillPath(path, brush)
```

I then had the determinant parallelogram for positive and negative determinants.

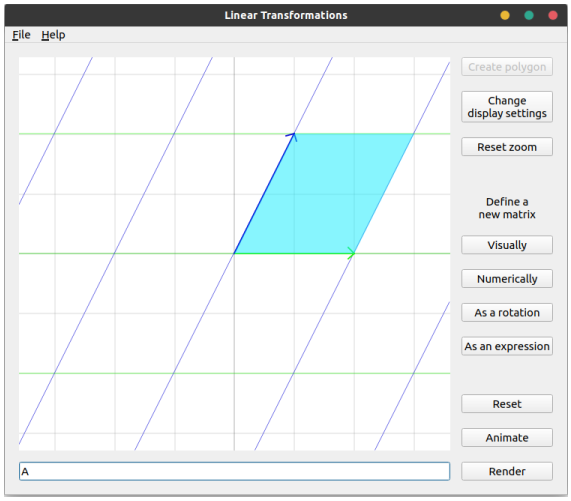


Figure 3.18: The blue parallelogram

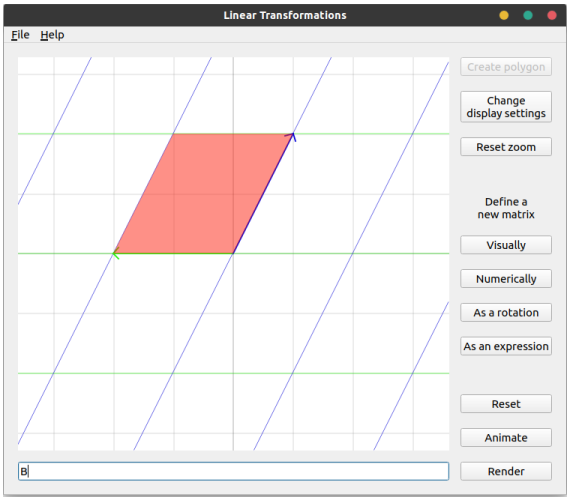


Figure 3.19: The red parallelogram

3.5.5 Adding the determinant text

Seeing the determinant as a shape is one thing, but knowing its exact value is also often very useful. To do this, I had to add a variable in the `DisplaySettings` for it, add a checkbox in the `DisplaySettingsDialog`, and create a method to actually draw the text in the right place, which I can call from `paintEvent()`.

```
# e344e50eccfd87c0834cfbdf459f0dd1d555fcd6
# src/lintrans/gui/settings.py

9   class DisplaySettings:
...
35      draw_determinant_text: bool = True
36      """This controls whether we should write the text value of the determinant inside the parallelogram.
37
38      The text only gets draw if :attr:`draw_determinant_parallelogram` is also True.
39      """


# e344e50eccfd87c0834cfbdf459f0dd1d555fcd6
# src/lintrans/gui/dialogs/settings.py
```

```
63      class DisplaySettingsDialog(SettingsDialog):
...
66          def __init__(self, display_settings: DisplaySettings, *args, **kwargs):
...
108             self.checkbox_draw_determinant_text = QCheckBox(self)
109             self.checkbox_draw_determinant_text.setText('Draw determinant text')
110             self.checkbox_draw_determinant_text.setToolTip(
111                 'Write the text value of the determinant inside the parallelogram'
112             )
```

```
        # e344e50eccfd87c0834cfbdf459f0dd1d555fcd6
        # src/lintrans/gui/plots/classes.py

142     class VectorGridPlot(BackgroundPlot):
...
416         def draw_determinant_text(self, painter: QPainter) -> None:
417             """Write the string value of the determinant in the middle of the parallelogram."""
418             painter.setPen(QPen(QColor(0, 0, 0), self.width_vector_line))
419             painter.drawText(
420                 *self.canvas_coords(
421                     (self.point_i[0] + self.point_j[0]) / 2,
422                     (self.point_i[1] + self.point_j[1]) / 2
423                 ),
424                 f'{self.det:.2f}'
425             )
```

It doesn't make much sense to show the text without also showing the parallelogram, so we should only show the text when the parallelogram is also being show, and the checkbox for the text should only be clickable when the parallelogram is enabled.

To do this, I created an `update_gui()` method which gets called when the parallelogram checkbox is clicked. This method will enable or disable the text checkbox appropriately.

```
        # e344e50eccfd87c0834cfbdf459f0dd1d555fcd6
        # src/lintrans/gui/plots/widgets.py

13      class VisualizeTransformationWidget(VectorGridPlot):
...
42          def paintEvent(self, event: QPaintEvent) -> None:
...
58              if self.display_settings.draw_determinant_parallelogram:
59                  self.draw_determinant_parallelogram(painter)
60
61              if self.display_settings.draw_determinant_text:
62                  self.draw_determinant_text(painter)
```

```
        # 517773e1ace0dc4485c425134cd36ba482ba65df
        # src/lintrans/gui/dialogs/settings.py

63      class DisplaySettingsDialog(SettingsDialog):
...
66          def __init__(self, display_settings: DisplaySettings, *args, **kwargs):
...
107             self.checkbox_draw_determinant_parallelogram.clicked.connect(self.update_gui)
...
173         def update_gui(self) -> None:
174             """Update the GUI according to other widgets in the GUI.
175
176             For example, this method updates which checkboxes are enabled based on the values of other checkboxes.
177             """
178             self.checkbox_draw_determinant_text.setEnabled(self.checkbox_draw_determinant_parallelogram.isChecked())
```

3.6 Fixing bugs and adding polish

3.6.1 Fixing an animation crash

The scaling logic in 3.3.6 creates a matrix $\mathbf{A}$ which is the start matrix plus some proportion of the difference between the target and start matrices. It then defines matrix $\mathbf{B}$ to be the matrix A normalised to have a determinant of 1. We then divide by $\det(\mathbf{B})$ to get matrix $\mathbf{C}$, which we then render.

This works very well for most matrices, but if we're animating from $\mathbf{I}$ to $-\mathbf{I}$ for example, then we can get the following problem:

When we're halfway through the animation, $p = \frac{1}{2}$.

$$\begin{aligned}\mathbf{A} &= \mathbf{S} + p(\mathbf{T} - \mathbf{S}) \\ &= \mathbf{I} + \frac{1}{2}(-\mathbf{I} - \mathbf{I}) \\ &= \mathbf{I} + \frac{-1}{\cancel{2}}\cancel{2}\mathbf{I} \\ &= \mathbf{I} - \mathbf{I} = \mathbf{0}\end{aligned}$$

I'm using $\mathbf{I}$ as an example here, but this can happen with the right p for many matrix pairs. Since $\mathbf{A} = \mathbf{0}$, $\det(\mathbf{A}) = 0$. We check for this case already when we find c:

```
# f7a91cdc35695f8fb9269b17bc103e42578072bd
# src/lintrans/gui/main_window.py
```

```
367                 if det_a == 0:
368                     c = 0
369                 else:
370                     c = np.sqrt(abs(det_start / det_a))
```

But if $\det(\mathbf{A}) = 0$, then $c = 0$ and $\det(\mathbf{B}) = 0$, so we also need to check that before we divide by it.

Old:

```
# f7a91cdc35695f8fb9269b17bc103e42578072bd
# src/lintrans/gui/main_window.py
```

```
383                 scalar = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
384                 matrix_to_render = scalar * matrix_b
```

New:

```
# 4383808a4cc29d192c55aca56161d8affda8c9a7
# src/lintrans/gui/main_window.py
```

```
384                 # That is all of course, if we can do that
385                 # We'll crash if we try to do this with det(B) == 0
386                 if det_b != 0:
387                     scalar = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
388                     matrix_to_render = scalar * matrix_b
389
390                 else:
391                     matrix_to_render = matrix_a
```

This change fixes a division by zero bug, which eliminates a possible crash here.

3.6.2 Limiting parallel lines

If you try to render a matrix like `0.01Irot(45)`, then the app ends up drawing as many parallel lines as it can physically fit in the viewport. This leads to a lot of lag, especially when zoomed out far. To fix this, I just introduced a maximum number of parallel lines. I chose 150 as a number that was big enough to have enough parallel lines for matrices that need a lot, while also causing virtually no lag.

```
# bd9aaa2e3037214f65d0fc1d12d67db35af0e5ec
# src/lintrans/gui/plots/classes.py

142 class VectorGridPlot(BackgroundPlot):
...
151     def __init__(self, *args, **kwargs):
...
169         self.max_parallel_lines = 150
```

```
# bd9aaa2e3037214f65d0fc1d12d67db35af0e5ec
# src/lintrans/gui/plots/classes.py

142 class VectorGridPlot(BackgroundPlot):
...
191     def draw_parallel_lines(self, painter: QPainter, vector: tuple[float, float], point: tuple[float, float]) ->
        ↪ None:
...
230         # Draw vertical lines
231         elif abs(vector_x) < 1e-12:
232             painter.drawLine(self.canvas_x(0), 0, self.canvas_x(0), self.height())
233
234             for i in range(max(abs(int(max_x / point_x)), self.max_parallel_lines)):
235                 painter.drawLine(
236                     self.canvas_x((i + 1) * point_x),
237                     0,
238                     self.canvas_x((i + 1) * point_x),
239                     self.height()
240                 )
241                 painter.drawLine(
242                     self.canvas_x(-1 * (i + 1) * point_x),
243                     0,
244                     self.canvas_x(-1 * (i + 1) * point_x),
245                     self.height()
246                 )
247
248         # Draw horizontal lines
249         elif abs(vector_y) < 1e-12:
250             painter.drawLine(0, self.canvas_y(0), self.width(), self.canvas_y(0))
251
252             for i in range(max(abs(int(max_y / point_y)), self.max_parallel_lines)):
253                 painter.drawLine(
254                     0,
255                     self.canvas_y((i + 1) * point_y),
256                     self.width(),
257                     self.canvas_y((i + 1) * point_y)
258                 )
259                 painter.drawLine(
260                     0,
261                     self.canvas_y(-1 * (i + 1) * point_y),
262                     self.width(),
263                     self.canvas_y(-1 * (i + 1) * point_y)
264                 )
265
266         # If the line is oblique, then we can use y = mx + c
267         else:
268             m = vector_y / vector_x
269             c = point_y - m * point_x
270
271             self.draw_oblique_line(painter, m, 0)
272
273             # We don't want to overshoot the max number of parallel lines,
274             # but we should also stop looping as soon as we can't draw any more lines
```

```
275                 for i in range(1, self.max_parallel_lines + 1):
276                     if not self.draw_pair_of_oblique_lines(painter, m, i * c):
277                         break
```

The idea behind this code is just to limit the maximum number of parallel lines that get drawn. It works perfectly for oblique lines, but there's a small bug for orthogonal lines that I never noticed. I just forgot to test it.

On lines 234 and 252, I call the built-in `max()` function with the maximum number of parallel lines and the total number of lines that could fit in the viewport. This should be a call to `min()` instead. I fixed this before releasing it for my end users, but it took an embarrassingly long time to notice something this simple.

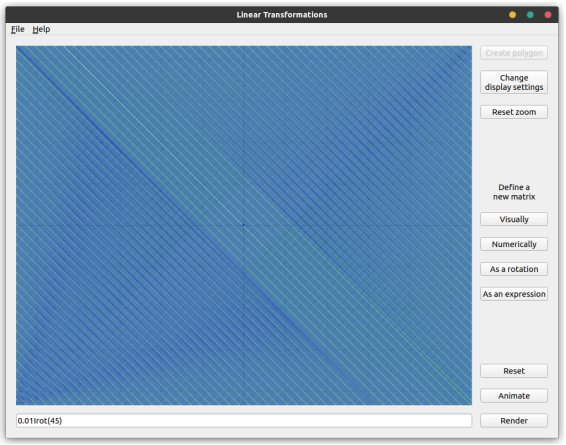


Figure 3.20: The old version with too many parallel lines.

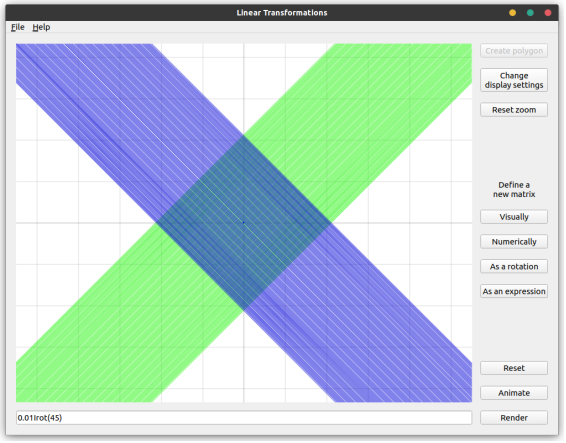


Figure 3.21: The fixed version with a maximum number of parallel lines.

3.6.3 Giving focus to the expression box

It would be quite nice to be able to just start typing an expression after defining a matrix or changing display settings. To do this, we can simply set the app's focus on the expression box after either of these actions.

Additionally, it would be nice to update the render buttons at the same time. That would allow the user to use a matrix in an expression, then define it, and be able to render the expression as soon as they close the dialog (assuming the expression is valid).

```
# bd7f8ba18266a8a095549d815dcfe6f24de514b6
# src/lintrans/gui/main_window.py

27  class LintransMainWindow(QMainWindow):
...
438     def assign_matrix_wrapper(self, matrix_wrapper: MatrixWrapper) -> None:
439         """Assign a new value to ``self.matrix_wrapper`` and give the expression box focus.
440
441         :param matrix_wrapper: The new value of the matrix wrapper to assign
442         :type matrix_wrapper: MatrixWrapper
443         """
444         self.matrix_wrapper = matrix_wrapper
445         self.lineedit_expression_box.setFocus()
446         self.update_render_buttons()
...
454     def assign_display_settings(self, display_settings: DisplaySettings) -> None:
455         """Assign a new value to ``self.plot.display_settings`` and give the expression box focus."""
456         self.plot.display_settings = display_settings
457         self.plot.update()
```

```
458             self.lineedit_expression_box.setFocus()
459             self.update_render_buttons()
```

3.6.4 Fixing a crash when animating singular matrices in sequence

If we have a matrix **A** defined as $\begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix}$, then when we try to render `A^-1`, we get a pop-up dialog box saying that we can't take the inverse of a singular matrix. This is good, since if NumPy just took the inverse blindly, it would crash. When we try to animate `A^-1`, we get the same pop-up box. When we try to use it in an animation sequence, however, like `rot(45),A^-1`, we don't check if each element of the sequence for singularity, so NumPy takes the inverse blindly and the whole app crashes. This is bad.

To fix this, we can simply catch the error when trying to evaluate the element in the sequence.

```
# 8db0df1d9d6a1be1f15a6f705e779d982db9ee29
# src/lintrans/gui/main_window.py

27  class LintransMainWindow(QMainWindow):
...
287     def animate_expression(self) -> None:
...
300         if ',' in text:
301             current_matrix = matrix_start
302
303             # For each expression in the list, right multiply it by the current matrix,
304             # and animate from the current matrix to that new matrix
305             for expr in text.split(',')[::-1]:
306                 try:
307                     new_matrix = self.matrix_wrapper.evaluate_expression(expr) @ current_matrix
308                 except linalg.LinAlgError:
309                     self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
310                     return
```

3.6.5 Allowing animations to be cancelled

Currently, if you try to reset the viewport partway through an animation, it just resets the basis vectors for a tick, but then they start moving again, because the animation loop is still running. To fix this, we can track whether we should be animating using an instance variable, set it to false when the user hits reset, and break out of the animation loop when it's false.

```
# b665bc59ec99664ed7b2c17f94e76ae49c6eb331
# src/lintrans/gui/main_window.py

27  class LintransMainWindow(QMainWindow):
...
33      def __init__(self):
...
45          self.animating: bool = False
46          self.animating_sequence: bool = False
...
269     def reset_transformation(self) -> None:
270         """Reset the visualized transformation back to the identity."""
271         self.plot.visualize_matrix_transformation(self.matrix_wrapper['I'])
272         self.animating = False
273         self.animating_sequence = False
274         self.plot.update()
...
292     def animate_expression(self) -> None:
...
304         # If there's commas in the expression, then we want to animate each part at a time
305         if ',' in text:
306             current_matrix = matrix_start
```

```
307                 self.animating_sequence = True
308
309                 # For each expression in the list, right multiply it by the current matrix,
310                 # and animate from the current matrix to that new matrix
311                 for expr in text.split(',')[::-1]:
312                     try:
313                         new_matrix = self.matrix_wrapper.evaluate_expression(expr) @ current_matrix
314                     except linalg.LinAlgError:
315                         self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
316                         return
317
318                     if not self.animating_sequence:
319                         break
320
321                     self.animate_between_matrices(current_matrix, new_matrix)
322                     current_matrix = new_matrix
323
324                     # Here we just redraw and allow for other events to be handled while we pause
325                     self.plot.update()
326                     QApplication.processEvents()
327                     QThread.msleep(self.plot.display_settings.animation_pause_length)
328
329                 self.animating_sequence = False
...
360         def animate_between_matrices(self, matrix_start: MatrixType, matrix_target: MatrixType, steps: int = 100) ->
            ↪ None:
...
365             self.animating = True
366
367             for i in range(0, steps + 1):
368                 if not self.animating:
369                     break
370
...
429             self.animating = False
```

Here, `self.animating_sequence` is whether a sequence is being animated, and `self.animating` is whether an individual matrix is currently being animated. An individual matrix means a matrix on its own, or a single element in a sequence. That means that `self.animating` can be set and unset multiple times in a single sequence.

3.6.6 Validating expression input

The user can only render or animate an expression if it's actually valid, as discussed in §3.1.3, and the render and animate buttons will be greyed out if the expression is invalid. But they can still type anything into the box.

It was at this point that I learned about the `QValidator` class[32]. This class allows me to control what the user can actually type. Using the implementation below, they can only enter characters that are allowed in valid matrix expressions.

```
# f73575c017548d754e4171449344a52cb44b7ef4
# src/lintrans/gui/main_window.py

28  class LintransMainWindow(QMainWindow):
...
34      def __init__(self):
...
125         self.lineedit_expression_box.setValidator(MatrixExpressionValidator(self))
```

```
# f73575c017548d754e4171449344a52cb44b7ef4
# src/lintrans/gui/validate.py

1   """This simple module provides a :class:`MatrixExpressionValidator` class to validate matrix expression input."""
2
```

```
3   from __future__ import annotations
4
5   import re
6
7   from PyQt5.QtGui import QValidator
8
9   from lintrans.matrices import parse
10
11
12  class MatrixExpressionValidator(QValidator):
13      """This class validates matrix expressions in an Qt input box."""
14
15      def validate(self, text: str, pos: int) -> tuple[QValidator.State, str, int]:
16          """Validate the given text according to the rules defined in the :mod:`lintrans.matrices` module."""
17          clean_text = re.sub(r'[\sA-Z\d.rot()^{},+-]', '', text)
18
19          if clean_text == '':
20              if parse.validate_matrix_expression(clean_text):
21                  return QValidator.Acceptable, text, pos
22              else:
23                  return QValidator.Intermediate, text, pos
24
25          return QValidator.Invalid, text, pos
26
```

I also then added validators to the definition dialogs, to make sure that users can only enter valid input. Qt5 provides some basic validators already, for things like integers and floating point numbers (called `double` in C++, equivalent to `float` in Python).

```
# a2fd14b99fa752a18b42352a01142ffbc2600570
# src/lintrans/gui/dialogs/define_new_matrix.py

213 class DefineNumericallyDialog(DefineDialog):
...
225         # tl = top left, br = bottom right, etc.
226         self.element_tl = QtWidgets.QLineEdit(self)
227         self.element_tl.textChanged.connect(self.update_confirm_button)
228         self.element_tl.setValidator(QDoubleValidator())
229
230         self.element_tr = QtWidgets.QLineEdit(self)
231         self.element_tr.textChanged.connect(self.update_confirm_button)
232         self.element_tr.setValidator(QDoubleValidator())
233
234         self.element_bl = QtWidgets.QLineEdit(self)
235         self.element_bl.textChanged.connect(self.update_confirm_button)
236         self.element_bl.setValidator(QDoubleValidator())
237
238         self.element_br = QtWidgets.QLineEdit(self)
239         self.element_br.textChanged.connect(self.update_confirm_button)
240         self.element_br.setValidator(QDoubleValidator())
...
299 class DefineAsARotationDialog(DefineDialog):
...
314         self.lineedit_angle = QtWidgets.QLineEdit(self)
315         self.lineedit_angle.setPlaceholderText('angle')
316         self.lineedit_angle.textChanged.connect(self.update_confirm_button)
317         self.lineedit_angle.setValidator(QDoubleValidator())
...
358 class DefineAsAnExpressionDialog(DefineDialog):
...
372         self.lineedit_expression_box = QtWidgets.QLineEdit(self)
373         self.lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
374         self.lineedit_expression_box.textChanged.connect(self.update_confirm_button)
375         self.lineedit_expression_box.setValidator(MatrixExpressionValidator())
```

3.6.7 Adding keyboard shortcuts

Keyboard shortcuts are often very useful and can make the process of using software much more efficient if you get good at using the shortcuts. On this note, I decided to add keyboard shortcuts to the display settings dialog.

Qt5 lets you use a `&` character in the text of a widget to act on the letter following it. This letter becomes underlined in the text, and the user can hold `Alt` and press this letter to activate the widget. I also want to be able to toggle the checkboxes by just pressing the letter without holding `Alt`, so I had to implement this myself with a dictionary and custom override of `keyPressEvent()`.

```
# 67d43a364ee2605b95b8caca9f1e4eb714cbb7c6
# src/lintrans/gui/dialogs/settings.py

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

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

        # 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_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_draw_determinant_text = QCheckBox(self)
        self.checkbox_draw_determinant_text.setText('Draw determinant &text')
        self.checkbox_draw_determinant_text.setToolTip(
```

```
119                 'Write the text value of the determinant inside the parallelogram'
120             )
121             self.dict_checkboxes['t'] = self.checkbox_draw_determinant_text
122 
123             # === Arrange the widgets in QGroupBoxes
124 
125             # Animations
126 
127             self.hlay_animation_pause_length = QHBoxLayout()
128             self.hlay_animation_pause_length.addWidget(self.label_animation_pause_length)
129             self.hlay_animation_pause_length.addWidget(self.lineedit_animation_pause_length)
130 
131             self.vlay_groupbox_animations = QVBoxLayout()
132             self.vlay_groupbox_animations.setSpacing(20)
133             self.vlay_groupbox_animations.addWidget(self.checkbox_smoothen_determinant)
134             self.vlay_groupbox_animations.addWidget(self.checkbox_applicative_animation)
135             self.vlay_groupbox_animations.addLayout(self.hlay_animation_pause_length)
136 
137             self.groupbox_animations = QGroupBox('Animations', self)
138             self.groupbox_animations.setLayout(self.vlay_groupbox_animations)
139 
140             # Matrix info
141 
142             self.vlay_groupbox_matrix_info = QVBoxLayout()
143             self.vlay_groupbox_matrix_info.setSpacing(20)
144             self.vlay_groupbox_matrix_info.addWidget(self.checkbox_draw_determinant_parallelogram)
145             self.vlay_groupbox_matrix_info.addWidget(self.checkbox_draw_determinant_text)
146 
147             self.groupbox_matrix_info = QGroupBox('Matrix info', self)
148             self.groupbox_matrix_info.setLayout(self.vlay_groupbox_matrix_info)
149 
150             self.vlay_options.addWidget(self.groupbox_animations)
151             self.vlay_options.addWidget(self.groupbox_matrix_info)
152 
153             # Finally, we load the current settings and update the GUI
154             self.load_settings()
155             self.update_gui()
...
188         def keyPressEvent(self, event: QKeyEvent) -> None:
189             """Handle a ``QKeyEvent`` by manually activating toggling checkboxes.
190 
191             Qt handles these shortcuts automatically and allows the user to do ``Alt + Key``
192             to activate a simple shortcut defined with ``&``. However, I like to be able to
193             just hit ``Key`` and have the shortcut activate.
194             """
195             letter = event.text().lower()
196             key = event.key()
197 
198             if letter in self.dict_checkboxes:
199                 self.dict_checkboxes[letter].animateClick()
200 
201             # Return or keypad enter
202             elif key == 0x01000004 or key == 0x01000005:
203                 self.button_confirm.click()
204 
205             # Escape
206             elif key == 0x01000000:
207                 self.button_cancel.click()
208 
209             else:
210                 event.ignore()
```

3.6.8 Centering text in the determinant parallelogram

The text in the determinant parallelogram is the numerical value of the determinant. Currently, it's not centered. It's drawn by just writing the text at a point, chosen to be the centre of the parallelogram. The `QPainter` class uses this point as the start of the baseline of the text, so it's effectively the bottom left corner.

```
# 67d43a364ee2605b95b8caca9f1e4eb714cbb7c6
# src/lintrans/gui/plots/classes.py

142 class VectorGridPlot(BackgroundPlot):
...
419     def draw_determinant_text(self, painter: QPainter) -> None:
420         """Write the string value of the determinant in the middle of the parallelogram."""
421         painter.setPen(QPen(QColor(0, 0, 0), self.width_vector_line))
422         painter.drawText(
423             *self.canvas_coords(
424                 (self.point_i[0] + self.point_j[0]) / 2,
425                 (self.point_i[1] + self.point_j[1]) / 2
426             ),
427             f'{self.det:.2f}'
428         )
```

Obviously, this text will look better if it's centered. To do this, we can create a bounding rectangle around the parallelogram and get the painter to draw the text in the centre of that rectangle.

We build the rectangle by getting the coordinates of each vertex of the parallelogram. Then the top left corner is the minimum x coordinate with the maximum y coordinate, and the bottom right corner is the maximum x with the minimum y.

```
# 9550416c0b273b16c90eb8d6319f5e17493ef9a8
# src/lintrans/gui/plots/classes.py

142 class VectorGridPlot(BackgroundPlot):
...
419     def draw_determinant_text(self, painter: QPainter) -> None:
420         """Write the string value of the determinant in the middle of the parallelogram."""
421         painter.setPen(QPen(QColor(0, 0, 0), self.width_vector_line))
422
423         # We're building a QRect that encloses the determinant parallelogram
424         # Then we can center the text in this QRect
425         coords: list[tuple[float, float]] = [
426             (0, 0),
427             self.point_i,
428             self.point_j,
429             (
430                 self.point_i[0] + self.point_j[0],
431                 self.point_i[1] + self.point_j[1]
432             )
433         ]
434
435         xs = [t[0] for t in coords]
436         ys = [t[1] for t in coords]
437
438         top_left = QPoint(*self.canvas_coords(min(xs), max(ys)))
439         bottom_right = QPoint(*self.canvas_coords(max(xs), min(ys)))
440
441         rect = QRectF(top_left, bottom_right)
442
443         painter.drawText(
444             rect,
445             Qt.AlignHCenter | Qt.AlignVCenter,
446             f'{self.det:.2f}'
447         )
```

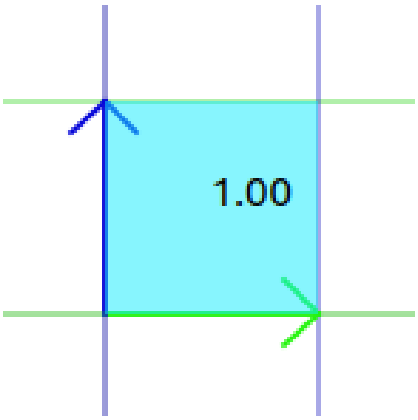


Figure 3.22: Text not centered.

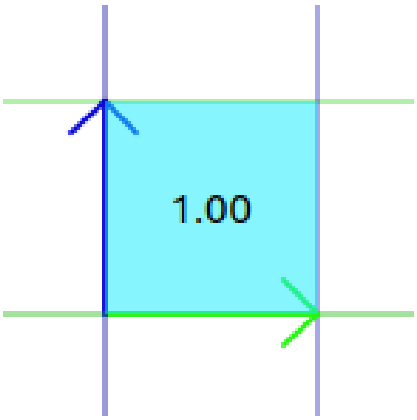


Figure 3.23: Text centered.

3.6.9 Defining matrices as expressions

Currently, you can "define" a matrix in terms of an expression, but it doesn't really define the matrix like that. Instead, it evaluates the expression immediately, and assigns that numerical result to the name you specified. It would be much better if the matrix could be actually defined as the expression, and then evaluated only when it needs to be used. Then, the user could have a matrix **M** defined as something like `0.5A^-1rot(45)B`, and it would always have that value, even if the user has changed the definition of **A** or **B** since defining **M**.

To do this, I'll have to completely change how matrices are stored and retrieved. The `MatrixWrapper` class contains a dictionary `self._matrices`, which currently maps `str` to `Optional[MatrixType]`, meaning that a matrix could be a 2×2 NumPy array, or nothing. I'm going to change this type to `Optional[Union[MatrixType, str]]`. This means that if a matrix exists, then it's either a 2×2 NumPy array or a string. We then check which one it is when we retrieve the matrix, and act accordingly. If it's an expression, then we evaluate and return the numerical result.

Here's the relevant parts of the old `MatrixWrapper` class:

```
# 9550416c0b273b16c90eb8d6319f5e17493ef9a8
# src/lintrans/matrices/wrapper.py

17  class MatrixWrapper:
...
35      def __init__(self):
36          """Initialise a :class:`MatrixWrapper` object with a dictionary of matrices which can be accessed."""
37          self._matrices: dict[str, Optional[MatrixType]] = {
38              'A': None, 'B': None, 'C': None, 'D': None,
39              'E': None, 'F': None, 'G': None, 'H': None,
40              'I': np.eye(2),  # I is always defined as the identity matrix
41              'J': None, 'K': None, 'L': None, 'M': None,
42              'N': None, 'O': None, 'P': None, 'Q': None,
43              'R': None, 'S': None, 'T': None, 'U': None,
44              'V': None, 'W': None, 'X': None, 'Y': None,
45              'Z': None
46          }
...
91      def __getitem__(self, name: str) -> Optional[MatrixType]:
92          """Get the matrix with the given name.
93
94          If it is a simple name, it will just be fetched from the dictionary. If the name is ``rot(x)``, with
95          a given angle in degrees, then we return a new matrix representing a rotation by that angle.
96
97          :param str name: The name of the matrix to get
98          :returns: The value of the matrix (may be None)
99          :rtype: Optional[MatrixType]
100
```

```
101             :raises NameError: If there is no matrix with the given name
102             """
103             # Return a new rotation matrix
104             if (match := re.match(r'rot\((-?\d*\.?\d*)\)', name)) is not None:
105                 return create_rotation_matrix(float(match.group(1)))
106 
107             if name not in self._matrices:
108                 raise NameError(f'Unrecognised matrix name "{name}"')
109 
110             # We copy the matrix before we return it so the user can't accidentally mutate the matrix
111             return copy(self._matrices[name])
112 
113         def __setitem__(self, name: str, new_matrix: Optional[MatrixType]) -> None:
114             """Set the value of matrix ``name`` with the new_matrix.
115 
116             :param str name: The name of the matrix to set the value of
117             :param Optional[MatrixType] new_matrix: The value of the new matrix (may be None)
118 
119             :raises NameError: If the name isn't a valid matrix name or is 'I'
120             :raises TypeError: If the matrix isn't a valid 2x2 NumPy array
121             """
122             if name not in self._matrices:
123                 raise NameError('Matrix name must be a single capital letter')
124 
125             if name == 'I':
126                 raise NameError('Matrix name cannot be "I"')
127 
128             if new_matrix is None:
129                 self._matrices[name] = None
130                 return
131 
132             if not is_matrix_type(new_matrix):
133                 raise TypeError('Matrix must be a 2x2 NumPy array')
134 
135             # All matrices must have float entries
136             a = float(new_matrix[0][0])
137             b = float(new_matrix[0][1])
138             c = float(new_matrix[1][0])
139             d = float(new_matrix[1][1])
140 
141             self._matrices[name] = np.array([[a, b], [c, d]])
142 
143         def is_valid_expression(self, expression: str) -> bool:
144             """Check if the given expression is valid, using the context of the wrapper.
145 
146             This method calls :func:`lintrans.matrices.parse.validate_matrix_expression`, but also
147             ensures that all the matrices in the expression are defined in the wrapper.
148 
149             :param str expression: The expression to validate
150             :returns: Whether the expression is valid in this wrapper
151             :rtype: bool
152             """
153             # Get rid of the transposes to check all capital letters
154             new_expression = expression.replace('^T', '').replace('^{T}', '')
155 
156             # Make sure all the referenced matrices are defined
157             for matrix in {x for x in new_expression if re.match('[A-Z]', x)}:
158                 if self[matrix] is None:
159                     return False
160 
161             return validate_matrix_expression(expression)
```

And here's the new version, which supports matrices defined as expressions:

```
# 01e866a74cf0f02ecba6438763d43e6eb90fe218
# src/lintrans/matrices/wrapper.py

17  class MatrixWrapper:
...
38      def __init__(self):
39          """Initialise a :class:`MatrixWrapper` object with a dictionary of matrices which can be accessed."""
```

```
40              self._matrices: dict[str, Optional[Union[MatrixType, str]]] = {
41                  'A': None, 'B': None, 'C': None, 'D': None,
42                  'E': None, 'F': None, 'G': None, 'H': None,
43                  'I': np.eye(2),  # I is always defined as the identity matrix
44                  'J': None, 'K': None, 'L': None, 'M': None,
45                  'N': None, 'O': None, 'P': None, 'Q': None,
46                  'R': None, 'S': None, 'T': None, 'U': None,
47                  'V': None, 'W': None, 'X': None, 'Y': None,
48                  'Z': None
49              }
...
94          def __getitem__(self, name: str) -> Optional[MatrixType]:
95              """Get the matrix with the given name.
96
97              If it is a simple name, it will just be fetched from the dictionary. If the name is ``rot(x)``, with
98              a given angle in degrees, then we return a new matrix representing a rotation by that angle.
99
100             :param str name: The name of the matrix to get
101             :returns: The value of the matrix (may be None)
102             :rtype: Optional[MatrixType]
103
104             :raises NameError: If there is no matrix with the given name
105             """
106             # Return a new rotation matrix
107             if (match := re.match(r'rot\((-?\d*\.?\d*)\)', name)) is not None:
108                 return create_rotation_matrix(float(match.group(1)))
109
110             if name not in self._matrices:
111                 raise NameError(f'Unrecognised matrix name "{name}"')
112
113             # We copy the matrix before we return it so the user can't accidentally mutate the matrix
114             matrix = copy(self._matrices[name])
115
116             if isinstance(matrix, str):
117                 return self.evaluate_expression(matrix)
118
119             return matrix
120
121         def __setitem__(self, name: str, new_matrix: Optional[Union[MatrixType, str]]) -> None:
122             """Set the value of matrix ``name`` with the new_matrix.
123
124             :param str name: The name of the matrix to set the value of
125             :param Optional[Union[MatrixType, str]] new_matrix: The value of the new matrix (may be None)
126
127             :raises NameError: If the name isn't a legal matrix name
128             :raises TypeError: If the matrix isn't a valid 2x2 NumPy array
129             """
130             if not (name in self._matrices and name != 'I'):
131                 raise NameError('Matrix name is illegal')
132
133             if new_matrix is None:
134                 self._matrices[name] = None
135                 return
136
137             if isinstance(new_matrix, str):
138                 if self.is_valid_expression(new_matrix):
139                     self._matrices[name] = new_matrix
140                     return
141
142             if not is_matrix_type(new_matrix):
143                 raise TypeError('Matrix must be a 2x2 NumPy array')
144
145             # All matrices must have float entries
146             a = float(new_matrix[0][0])
147             b = float(new_matrix[0][1])
148             c = float(new_matrix[1][0])
149             d = float(new_matrix[1][1])
150
151             self._matrices[name] = np.array([[a, b], [c, d]])
152
153         def get_expression(self, name: str) -> Optional[str]:
154             """If the named matrix is defined as an expression, return that expression, else return None.
155
```

```
156             :param str name: The name of the matrix
157             :returns: The expression that the matrix is defined as, or None
158             :rtype: Optional[str]
159
160             :raises NameError: If the name is invalid
161             """
162             if name not in self._matrices:
163                 raise NameError('Matrix must have a legal name')
164
165             matrix = self._matrices[name]
166             if isinstance(matrix, str):
167                 return matrix
168
169             return None
170
171         def is_valid_expression(self, expression: str) -> bool:
172             """Check if the given expression is valid, using the context of the wrapper.
173
174             This method calls :func:`lintrans.matrices.parse.validate_matrix_expression`, but also
175             ensures that all the matrices in the expression are defined in the wrapper.
176
177             :param str expression: The expression to validate
178             :returns: Whether the expression is valid in this wrapper
179             :rtype: bool
180             """
181             # Get rid of the transposes to check all capital letters
182             new_expression = expression.replace('^T', '').replace('^{T}', '')
183
184             # Make sure all the referenced matrices are defined
185             for matrix in {x for x in new_expression if re.match('[A-Z]', x)}:
186                 if self[matrix] is None:
187                     return False
188
189                 if (expr := self.get_expression(matrix)) is not None:
190                     if not self.is_valid_expression(expr):
191                         return False
192
193             return validate_matrix_expression(expression)
```

One of the more subtle things added here is on lines 189-191. When checking if an expression is valid in the context of the wrapper, we have to make sure all the referenced matrices are actually defined, but if any of those matrices are defined as an expression, then obviously that expression has to be valid as well. This recursion means that all references to matrices must be valid, even traversing down through matrices that are defined as expressions.

I also added some unit tests to automatically test this new feature.

```
# 239bcbfd1dde3f7623318d03e8544dd67dc02e3d
# tests/matrices/matrix_wrapper/test_setitem_and_getitem.py

42  def test_set_expression(test_wrapper: MatrixWrapper) -> None:
43      """Test that MatrixWrapper.__setitem__() can accept a valid expression."""
44      test_wrapper['N'] = 'A^2'
45      test_wrapper['O'] = 'BA+2C'
46      test_wrapper['P'] = 'E^T'
47      test_wrapper['Q'] = 'C^-1B'
48      test_wrapper['R'] = 'A^{2}3B'
49      test_wrapper['S'] = 'N^-1'
50      test_wrapper['T'] = 'PQP^-1'
51
52      with pytest.raises(TypeError):
53          test_wrapper['U'] = 'A+1'
54          test_wrapper['V'] = 'K'
55          test_wrapper['W'] = 'L^2'
56          test_wrapper['X'] = 'M^-1'
57
58
59  def test_simple_dynamic_evaluation(test_wrapper: MatrixWrapper) -> None:
60      """Test that expression-defined matrices are evaluated dynamically."""
61      test_wrapper['N'] = 'A^2'
```

```
62          test_wrapper['O'] = '4B'
63          test_wrapper['P'] = 'A+C'
64
65          assert (test_wrapper['N'] == test_wrapper.evaluate_expression('A^2')).all()
66          assert (test_wrapper['O'] == test_wrapper.evaluate_expression('4B')).all()
67          assert (test_wrapper['P'] == test_wrapper.evaluate_expression('A+C')).all()
68
69          assert (test_wrapper.evaluate_expression('N^2 + 3O') ==
70                  la.matrix_power(test_wrapper.evaluate_expression('A^2'), 2) +
71                  3 * test_wrapper.evaluate_expression('4B')
72                  ).all()
73          assert (test_wrapper.evaluate_expression('P^-1 - 3NO^2') ==
74                  la.inv(test_wrapper.evaluate_expression('A+C')) -
75                  (3 * test_wrapper.evaluate_expression('A^2')) @
76                  la.matrix_power(test_wrapper.evaluate_expression('4B'), 2)
77                  ).all()
78
79          test_wrapper['A'] = np.array([
80              [19, -21.5],
81              [84, 96.572]
82          ])
83          test_wrapper['B'] = np.array([
84              [-0.993, 2.52],
85              [1e10, 0]
86          ])
87          test_wrapper['C'] = np.array([
88              [0, 19512],
89              [1.414, 19]
90          ])
91
92          assert (test_wrapper['N'] == test_wrapper.evaluate_expression('A^2')).all()
93          assert (test_wrapper['O'] == test_wrapper.evaluate_expression('4B')).all()
94          assert (test_wrapper['P'] == test_wrapper.evaluate_expression('A+C')).all()
95
96          assert (test_wrapper.evaluate_expression('N^2 + 3O') ==
97                  la.matrix_power(test_wrapper.evaluate_expression('A^2'), 2) +
98                  3 * test_wrapper.evaluate_expression('4B')
99                  ).all()
100         assert (test_wrapper.evaluate_expression('P^-1 - 3NO^2') ==
101                 la.inv(test_wrapper.evaluate_expression('A+C')) -
102                 (3 * test_wrapper.evaluate_expression('A^2')) @
103                 la.matrix_power(test_wrapper.evaluate_expression('4B'), 2)
104                 ).all()
105
106
107     def test_recursive_dynamic_evaluation(test_wrapper: MatrixWrapper) -> None:
108         """Test that dynamic evaluation works recursively."""
109         test_wrapper['N'] = 'A^2'
110         test_wrapper['O'] = '4B'
111         test_wrapper['P'] = 'A+C'
112
113         test_wrapper['Q'] = 'N^-1'
114         test_wrapper['R'] = 'P-4O'
115         test_wrapper['S'] = 'NOP'
116
117         assert test_wrapper['Q'] == pytest.approx(test_wrapper.evaluate_expression('A^-2'))
118         assert test_wrapper['R'] == pytest.approx(test_wrapper.evaluate_expression('A + C - 16B'))
119         assert test_wrapper['S'] == pytest.approx(test_wrapper.evaluate_expression('A^{2}4BA + A^{2}4BC'))
120
121
122     def test_set_identity_error(new_wrapper: MatrixWrapper) -> None:
123         """Test that MatrixWrapper().__setitem__() raises a NameError when trying to assign to I."""
124         with pytest.raises(NameError):
125             new_wrapper['I'] = test_matrix
126
127
128     def test_set_name_error(new_wrapper: MatrixWrapper) -> None:
129         """Test that MatrixWrapper().__setitem__() raises a NameError when trying to assign to an invalid name."""
130         with pytest.raises(NameError):
131             new_wrapper['bad name'] = test_matrix
132             new_wrapper['123456'] = test_matrix
133             new_wrapper['Th15 Is an 1nV@l1D n@m3'] = test_matrix
134             new_wrapper['abc'] = test_matrix
```

```
135             new_wrapper['a'] = test_matrix
136
137
138     def test_set_type_error(new_wrapper: MatrixWrapper) -> None:
139         """Test that MatrixWrapper().__setitem__() raises a TypeError when trying to set a non-matrix."""
140         with pytest.raises(TypeError):
141             new_wrapper['M'] = 12
142             new_wrapper['M'] = [1, 2, 3, 4, 5]
143             new_wrapper['M'] = [[1, 2], [3, 4]]
144             new_wrapper['M'] = True
145             new_wrapper['M'] = 24.3222
146             new_wrapper['M'] = 'This is totally a matrix, I swear'
147             new_wrapper['M'] = MatrixWrapper
148             new_wrapper['M'] = MatrixWrapper()
149             new_wrapper['M'] = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
150             new_wrapper['M'] = np.eye(100)
```

```
# ea00703f19c13af86c39ae30170569819937fa31
# tests/matrices/matrix_wrapper/test_misc.py

1   """Test the miscellaneous methods of the MatrixWrapper class."""
2
3   from lintrans.matrices import MatrixWrapper
4
5
6   def test_get_expression(test_wrapper: MatrixWrapper) -> None:
7       """Test the get_expression method of the MatrixWrapper class."""
8       test_wrapper['N'] = 'A^2'
9       test_wrapper['O'] = '4B'
10      test_wrapper['P'] = 'A+C'
11
12      test_wrapper['Q'] = 'N^-1'
13      test_wrapper['R'] = 'P-4O'
14      test_wrapper['S'] = 'NOP'
15
16      assert test_wrapper.get_expression('A') is None
17      assert test_wrapper.get_expression('B') is None
18      assert test_wrapper.get_expression('C') is None
19      assert test_wrapper.get_expression('D') is None
20      assert test_wrapper.get_expression('E') is None
21      assert test_wrapper.get_expression('F') is None
22      assert test_wrapper.get_expression('G') is None
23
24      assert test_wrapper.get_expression('N') == 'A^2'
25      assert test_wrapper.get_expression('O') == '4B'
26      assert test_wrapper.get_expression('P') == 'A+C'
27
28      assert test_wrapper.get_expression('Q') == 'N^-1'
29      assert test_wrapper.get_expression('R') == 'P-4O'
30      assert test_wrapper.get_expression('S') == 'NOP'
```

I then had to fix a small bug where the `DefineAsAnExpressionDialog` would evaluate the expression before assigning it, so I had to change that to just assign the test instead.

```
# 54e10dbfd3a1f3a962955c7fa3908848f5bd95b0
# src/lintrans/gui/dialogs/define_new_matrix.py

343 class DefineAsAnExpressionDialog(DefineDialog):
...
388     def confirm_matrix(self) -> None:
389         """Evaluate the matrix expression and assign its value to the name in the combo box."""
390         self.matrix_wrapper[self.selected_letter] = self.lineedit_expression_box.text()
391         self.accept()
```

I also created a virtual method in the `DefineDialog` superclass, which standardised how dialogs load a matrix when it's selected in the drop-down. The numerical and visual definition dialogs already did this, but it was inconsistent, so I made it the same across all subclasses, and added it to the expression dialog.

```
# d1b60b20666ab9297cdbf675b6226587fd2e417f
# src/lintrans/gui/dialogs/define_new_matrix.py

class DefineDialog(QDialog):
...
    def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
...
        self.combobox_letter = QtWidgets.QComboBox(self)

        for letter in ALPHABET_NO_I:
            self.combobox_letter.addItem(letter)

        self.combobox_letter.activated.connect(self.load_matrix)
...
    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 combobox.changed signal in this constructor, rather than having to
        define that in the constructor of every subclass.
        """
...
class DefineAsAnExpressionDialog(DefineDialog):
...
    def load_matrix(self, index: int) -> None:
        """If the selected matrix is defined an expression, load that expression into the box."""
        name = ALPHABET_NO_I[index]

        if (expr := self.matrix_wrapper.get_expression(name)) is not None:
            self.lineedit_expression_box.setText(expr)
        else:
            self.lineedit_expression_box.setText('')
```

Unfortunately, my initial implementation of this had a few bugs, and I noticed a few hours later that if you first define **A** as anything concrete, then you can define **A** to be the expression `A`. Then, when you put it in the expression box, the app just crashes. This is because it recurs forever, since it doesn't realise that the definition of **A** is self-referential[14].

To fix this, I can check that the expression is valid and that it doesn't contain itself before assigning the expression to the matrix name.

```
# 742e0955e344deab2c9302ba9a6c7298ec4583d4
# src/lintrans/gui/dialogs/define_new_matrix.py

class DefineAsAnExpressionDialog(DefineDialog):
...
    def update_confirm_button(self) -> None:
...
        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)
```

I also added this logic directly to the wrapper, so that there was no risk of me creating this kind of bug elsewhere.

```
# e56a5a90034f8335b046dd1bf76321eb48892050
# src/lintrans/matrices/wrapper.py

class MatrixWrapper:
...
    def __setitem__(self, name: str, new_matrix: Optional[Union[MatrixType, str]]) -> None:
```

[14] Obviously it doesn't actually recur forever, but Python stops recursion after 1000 levels and crashes the program.

```
...
145             if isinstance(new_matrix, str):
146                 if self.is_valid_expression(new_matrix):
147                     if name not in new_matrix:
148                         self._matrices[name] = new_matrix
149                         return
150                     else:
151                         raise ValueError('Cannot define a matrix recursively')
```

While I was working with expressions so much, I realised that defining a matrix as a rotation was a bit redundant when you can just use an expression like `rot(45)`. I spoke to the teacher that's going to use `lintrans` when it's finished, and she said that radians aren't really needed. The radians checkbox was the only unique part of the `DefineAsARotationDialog` class. Since it's not important, I decided to remove the whole dialog.

3.7 Implementing eigenstuffs

It's not universal, but the word 'eigenstuffs' is common enough in mathematics that I'm comfortable using it here to mean eigenvalues, eigenvectors, and eigenlines, where an eigenline is the span of an eigenvector.

3.7.1 Drawing eigenvectors

An eigenvector $\mathbf{v}$ of a matrix $\mathbf{M}$ is a vector that satisfies the equation $\mathbf{Mv} = \lambda\mathbf{v}$ for some scalar λ. Thankfully, I don't have to worry about actually computing $\mathbf{v}$ or λ, since NumPy has the `numpy.linalg.eig()` function[22].

This function takes a square matrix and returns an array of eigenvalues (λ), and a matrix of their associated eigenvectors ($\mathbf{v}$). Some matrices don't have any real eigenvalues, but all 2×2 matrices will have 2 (possibly complex) eigenvalues, as a direct consequence of the Fundamental Theorem of Algebra[15]. We don't want to try to render an eigenvector if its eigenvalue is complex, so we have to check and only render the real ones. Python doesn't distinguish really between `float` and `complex` types, so we can just check the `.imag` property no matter what. If it's 0, then we keep the eigenvalue. NumPy normalizes the eigenvectors to have a length of 1, but I'd much prefer them to have a length equal to their associated eigenvalue. To do that, we can just multiply the eigenvectors by their eigenvalues.

We then just draw a vector, consisting of a line and an arrowhead, from the origin to the extended eigenvector.

```
# b8614334de5cba4b1a6d92508b08fa8bd2fe77c0
# src/lintrans/gui/plots/classes.py

class VectorGridPlot(BackgroundPlot):
...
    def __init__(self, *args, **kwargs):
...
        self.colour_eigen = QColor('#fff900')
...
    def draw_eigenvectors(self, painter: QPainter) -> None:
        """Draw the eigenvectors of the displayed matrix transformation."""
        painter.setPen(QPen(self.colour_eigen, self.width_vector_line))

        values, vectors = np.linalg.eig(self.matrix)
        vectors = vectors.T

        for value, vector in zip(values, vectors):
            x = value * vector[0]
            y = value * vector[1]

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

            painter.drawLine(*self.canvas_origin, *self.canvas_coords(x, y))
            self.draw_arrowhead_away_from_origin(painter, (x, y))
```

```
# b8614334de5cba4b1a6d92508b08fa8bd2fe77c0
# src/lintrans/gui/plots/widgets.py

class VisualizeTransformationWidget(VectorGridPlot):
...
    def paintEvent(self, event: QPaintEvent) -> None:
...
        self.draw_eigenvectors(painter)
```

[15] $\mathbf{Mv} = \lambda\mathbf{v} \implies \mathbf{Mv} = \lambda\mathbf{Iv} \implies (\mathbf{M} - \lambda\mathbf{I})\mathbf{v} = \mathbf{0} \implies \det(\mathbf{M} - \lambda\mathbf{I}) = 0$ (since we only want non-zero vectors)
$\implies \begin{vmatrix} a-\lambda & b \\ c & d-\lambda \end{vmatrix} = 0 \implies (a-\lambda)(d-\lambda) - bc = 0 \implies \lambda^2 - (a+d)\lambda + (ad-bc) = 0$
$\implies \lambda$ has 2 solutions in $\mathbb{C}$ by the Fundamental Theorem of Algebra[12]

At this point in development, I didn't particularly care about the colours of various elements. It was more important to get things working first, so I ended up choosing this horrible yellow for the eigenvectors. It's clearly an awful choice for text, and it's not very good for the eigenvectors either, since it makes them hard to see against the white background. I wasn't really considering the usability features discussed in §2.4, but since I was the only user, and changing a few colours later on wouldn't be much work, I wasn't worried about it.

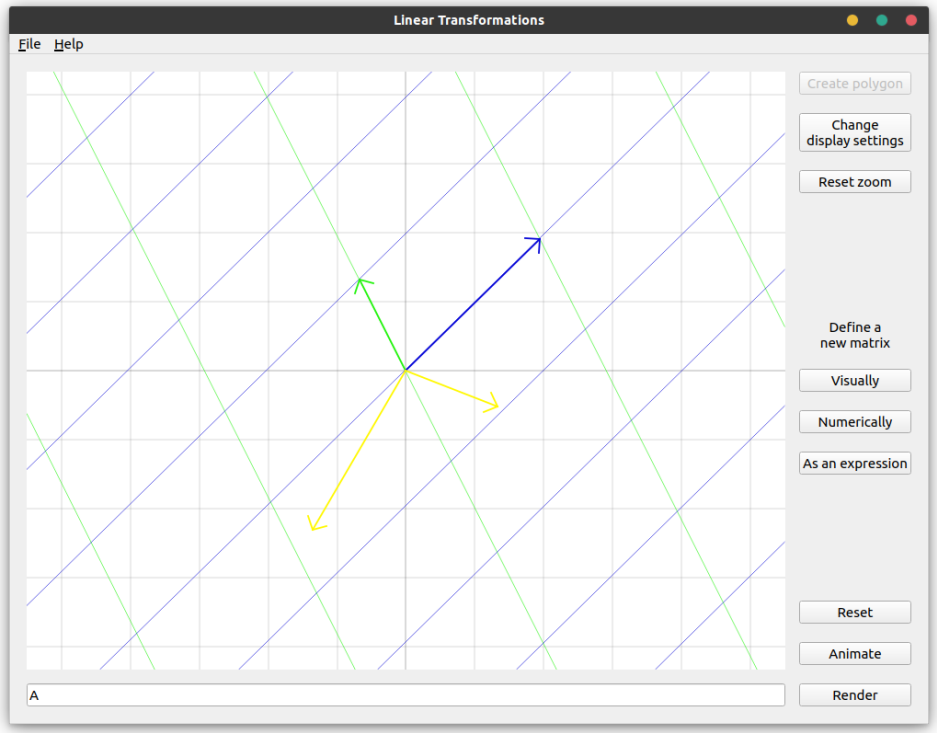


Figure 3.24: The eigenvectors being displayed for an arbitrary matrix

3.7.2 Adding display settings for eigenvectors

Once I'd got the eigenvectors working and being drawn, I wanted to be able to turn them on and off from the display settings. This was a simple case of just adding it to the `DisplaySettings` dataclass, adding the checkbox to the dialog, and only drawing the eigenvectors if this display setting is true.

```
# 3ebb5997a7a887e751b5f4f717aa5161ed013e62
# src/lintrans/gui/settings.py

 9  class DisplaySettings:
...
41      draw_eigenvectors: bool = False
42      """This controls whether we should draw the eigenvectors of the transformation."""


# 3ebb5997a7a887e751b5f4f717aa5161ed013e62
# src/lintrans/gui/dialogs/settings.py

63  class DisplaySettingsDialog(SettingsDialog):
...
66      def __init__(self, display_settings: DisplaySettings, *args, **kwargs):
...
123         self.checkbox_draw_eigenvectors = QCheckBox(self)
124         self.checkbox_draw_eigenvectors.setText('Draw &eigenvectors')
125         self.checkbox_draw_eigenvectors.setToolTip('Draw the eigenvectors of the transformations')
126         self.dict_checkboxes['e'] = self.checkbox_draw_eigenvectors
```

```
# 3ebb5997a7a887e751b5f4f717aa5161ed013e62
# src/lintrans/gui/plots/widgets.py

13  class VisualizeTransformationWidget(VectorGridPlot):
...
42      def paintEvent(self, event: QPaintEvent) -> None:
...
58          if self.display_settings.draw_eigenvectors:
59              self.draw_eigenvectors(painter)
```

3.7.3 Refactoring drawing vectors

Since I've now got several drawing methods that involve drawing vectors, I thought I should factor out this functionality into a separate utility method which I could call from all these places.

```
# 754eba0318a682b068a8a5d5ca451decbaa204ce
# src/lintrans/gui/plots/classes.py

142 class VectorGridPlot(BackgroundPlot):
...
388     def draw_position_vector(self, painter: QPainter, point: tuple[float, float], colour: QColor) -> None:
389         """Draw a vector from the origin to the given point.
390
391         :param QPainter painter: The ``QPainter`` object to use to draw the arrowheads with
392         :param point: The tip of the position vector in grid coords
393         :type point: tuple[float, float]
394         :param QColor colour: The colour to draw the position vector in
395         """
396         painter.setPen(QPen(colour, self.width_vector_line))
397         painter.drawLine(*self.canvas_origin, *self.canvas_coords(*point))
398         self.draw_arrowhead_away_from_origin(painter, point)
399
400     def draw_basis_vectors(self, painter: QPainter) -> None:
401         """Draw arrowheads at the tips of the basis vectors.
402
403         :param QPainter painter: The ``QPainter`` object to use to draw the arrowheads with
404         """
405         self.draw_position_vector(painter, self.point_i, self.colour_i)
406         self.draw_position_vector(painter, self.point_j, self.colour_j)
...
454     def draw_eigenvectors(self, painter: QPainter) -> None:
455         """Draw the eigenvectors of the displayed matrix transformation."""
456         values, vectors = np.linalg.eig(self.matrix)
457         vectors = vectors.T
458
459         for value, vector in zip(values, vectors):
460             x = value * vector[0]
461             y = value * vector[1]
462
463             if x.imag != 0 or y.imag != 0:
464                 continue
465
466             self.draw_position_vector(painter, (x, y), self.colour_eigen)
```

When I was testing this refactor, I realised that the `draw_transformed_grid()` method originally drew the bodies of the basis vectors and the caller had to draw the arrowheads separately. This is silly and was never planned that way; it was an unfortunate consequence of implementing the lines and arrowheads at different times. But now after this refactor, the caller has to call `draw_basis_vectors()` to draw the whole basis vectors. So I had to add this call to `VisualizeTransformationWidget.paintEvent()` and `DefineVisuallyWidget.paintEvent()` in `src/lintrans/gui/plots/widgets.py` and remove the calls to `draw_vector_arrowheads()`.

3.7.4 Adding eigenlines

Drawing some eigenvectors that point in a general direction is fine, but drawing the whole span of the vector would be much more useful. These spans are called eigenlines, and are just lines in the direction of the vector. To implement these, I knew I would have to get the eigenvalues and eigenvectors, zip them, and iterate over them like I did in §3.7.1. To make this simpler, I decided to factor out this zipping into a separate `self.eigs` property[16].

```
# e1606f1e45ba93102dddb74b45ab22649a63fa53
# src/lintrans/gui/plots/classes.py

144 class VectorGridPlot(BackgroundPlot):
...
187     @property
188     def eigs(self) -> Iterable[tuple[float, NDArray[(1, 2), Float]]]:
189         """Return the eigenvalues and eigenvectors zipped together to be iterated over.
190
191         :rtype: Iterable[tuple[float, NDArray[(1, 2), Float]]]
192         """
193         values, vectors = np.linalg.eig(self.matrix)
194         return zip(values, vectors.T)
...
465     def draw_eigenvectors(self, painter: QPainter) -> None:
466         """Draw the eigenvectors of the displayed matrix transformation."""
467         for value, vector in self.eigs:
468             x = value * vector[0]
469             y = value * vector[1]
470
471             if x.imag != 0 or y.imag != 0:
472                 continue
473
474             self.draw_position_vector(painter, (x, y), self.colour_eigen)
```

I could then create a new method to find the gradient of the vector line and draw an orthogonal or oblique line respectively. Like before, we only want to render the eigenlines for real-valued eigenvectors, so we have to check their imaginary part.

```
# 8d4d41fc4780cc037be39a0e574158e6cd34e997
# src/lintrans/gui/plots/classes.py

144 class VectorGridPlot(BackgroundPlot):
...
476     def draw_eigenlines(self, painter: QPainter) -> None:
477         """Draw the eigenlines (invariant lines).
478
479         :param QPainter painter: The painter to draw the lines with
480         """
481         painter.setPen(QPen(self.colour_eigen, self.width_transformed_grid))
482
483         for value, vector in self.eigs:
484             if value.imag != 0:
485                 continue
486
487             x, y = vector
488
489             if x == 0:
490                 x_mid = int(self.width() / 2)
491                 painter.drawLine(x_mid, 0, x_mid, self.height())
492
493             elif y == 0:
494                 y_mid = int(self.height() / 2)
495                 painter.drawLine(0, y_mid, self.width(), y_mid)
496
497             else:
498                 self.draw_oblique_line(painter, y / x, 0)
```

[16] A `@property` in Python is a value on a class which is dynamically evaluated only when it's needed. It functions just like a method with no arguments, but has more concise syntax for the caller.

I then just had to put these new eigenlines behind a display setting.

```
# 12cfabde606ebd3d48b2c3efaad0412f6100c3c5
# src/lintrans/gui/settings.py

 9  class DisplaySettings:
...
44      draw_eigenlines: bool = False
45      """This controls whether we should draw the eigenlines of the transformation."""


# 12cfabde606ebd3d48b2c3efaad0412f6100c3c5
# src/lintrans/gui/dialogs/settings.py

63  class DisplaySettingsDialog(SettingsDialog):
...
66      def __init__(self, display_settings: DisplaySettings, *args, **kwargs):
...
128         self.checkbox_draw_eigenlines = QCheckBox(self)
129         self.checkbox_draw_eigenlines.setText('Draw eigen&lines')
130         self.checkbox_draw_eigenlines.setToolTip('Draw the eigenlines (invariant lines) of the transformations')
131         self.dict_checkboxes['l'] = self.checkbox_draw_eigenlines


# 12cfabde606ebd3d48b2c3efaad0412f6100c3c5
# src/lintrans/gui/plots/widgets.py

13  class VisualizeTransformationWidget(VectorGridPlot):
...
42      def paintEvent(self, event: QPaintEvent) -> None:
...
58          if self.display_settings.draw_eigenlines:
59              self.draw_eigenlines(painter)
```

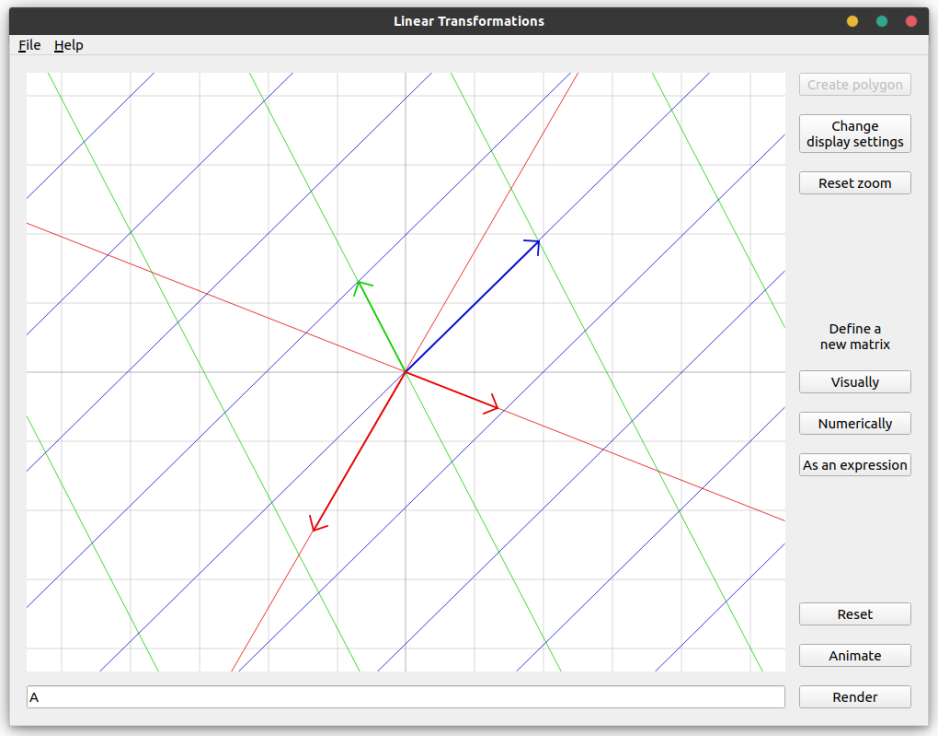


Figure 3.25: The eigenvectors being displayed for a similar matrix as in Figure 3.24

3.7.5 Adding eigenvalues as text

The next thing I wanted to do was tell the user what the actual eigenvalue numbers were. I decided the best way to do this would be to have the eigenvectors labelled with their associated eigenvalues.

To align the text properly, I decided to do something similar to what I did for determinants in §3.6.8. There, I enclosed the parallelogram in a rectangle and drew the text in the centre of it. Here, I decided to create a rectangle based on which quadrant the eigenvector was in. The rectangle would have corners of the tip of the eigenvector, and the appropriate corner of the viewport. I also had to choose a suitable alignment for each quadrant, so that the text would appear in the correct corner of the rectangle, next to the tip of the eigenvector.

```
# a7e9a17ecdb2a585e9f766b13029116d7ba29bdc
# src/lintrans/gui/plots/classes.py

144 class VectorGridPlot(BackgroundPlot):
...
466     def draw_eigenvectors(self, painter: QPainter) -> None:
467         """Draw the eigenvectors of the displayed matrix transformation."""
468         for value, vector in self.eigs:
469             x = value * vector[0]
470             y = value * vector[1]
471
472             if x.imag != 0 or y.imag != 0:
473                 continue
474
475             self.draw_position_vector(painter, (x, y), self.colour_eigen)
476
477             # Now we need to draw the eigenvalue at the tip of the eigenvector
478
479             offset = 3
480             top_left: QPoint
481             bottom_right: QPoint
482             alignment_flags: int
483
484             if x >= 0 and y >= 0:  # Q1
485                 top_left = QPoint(self.canvas_x(x) + offset, 0)
486                 bottom_right = QPoint(self.width(), self.canvas_y(y) - offset)
487                 alignment_flags = Qt.AlignLeft | Qt.AlignBottom
488
489             elif x < 0 and y >= 0:  # Q2
490                 top_left = QPoint(0, 0)
491                 bottom_right = QPoint(self.canvas_x(x) - offset, self.canvas_y(y) - offset)
492                 alignment_flags = Qt.AlignRight | Qt.AlignBottom
493
494             elif x < 0 and y < 0:  # Q3
495                 top_left = QPoint(0, self.canvas_y(y) + offset)
496                 bottom_right = QPoint(self.canvas_x(x) - offset, self.height())
497                 alignment_flags = Qt.AlignRight | Qt.AlignTop
498
499             else:  # Q4
500                 top_left = QPoint(self.canvas_x(x) + offset, self.canvas_y(y) + offset)
501                 bottom_right = QPoint(self.width(), self.height())
502                 alignment_flags = Qt.AlignLeft | Qt.AlignTop
503
504             painter.setPen(QPen(self.colour_text, self.width_vector_line))
505             painter.drawText(QRectF(top_left, bottom_right), alignment_flags, f'{value:.2f}')
```

3.7.6 A tiny UI change

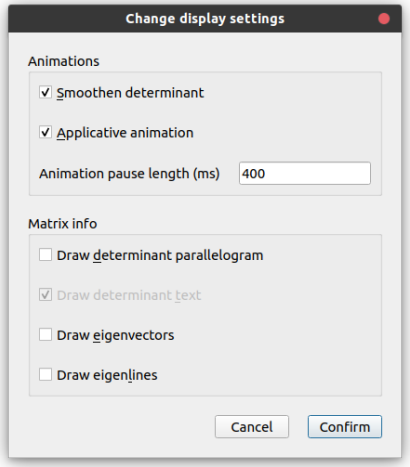


Figure 3.26: The display settings

This bit isn't really related to eigenvectors, but it's a tiny change that doesn't really have a good place anywhere else, and it fits roughly here chronologically.

I really liked the groupboxes used in the display settings (left) and I'd quite like to enclose the matrix definition buttons in their own groupbox to separate them from the rest of the UI and better associate them with the label above them. This was a trivial addition.

```
# 90425137edd4596219ab564ccbeccd65b5754008
# src/lintrans/gui/main_window.py

27  class LintransMainWindow(QMainWindow):
...
33      def __init__(self):
...
173         self.vlay_define_new_matrix = QVBoxLayout()
174         self.vlay_define_new_matrix.setSpacing(20)
175         self.vlay_define_new_matrix.addWidget(self.button_define_visually)
176         self.vlay_define_new_matrix.addWidget(self.button_define_numerically)
177         self.vlay_define_new_matrix.addWidget(self.button_define_as_expression)
178
179         self.groupbox_define_new_matrix = QtWidgets.QGroupBox('Define a new matrix', self)
180         self.groupbox_define_new_matrix.setLayout(self.vlay_define_new_matrix)
...
226         self.vlay_right.addWidget(self.groupbox_define_new_matrix)
```

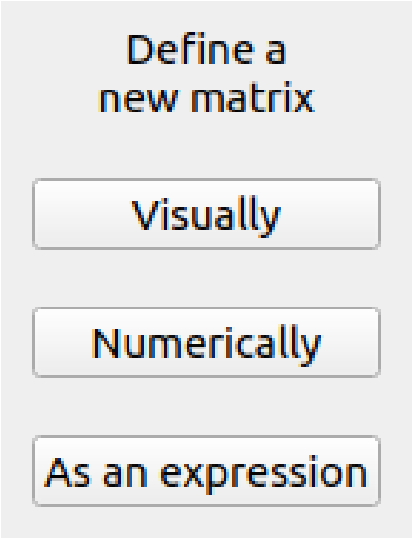


Figure 3.27: The old matrix definition buttons

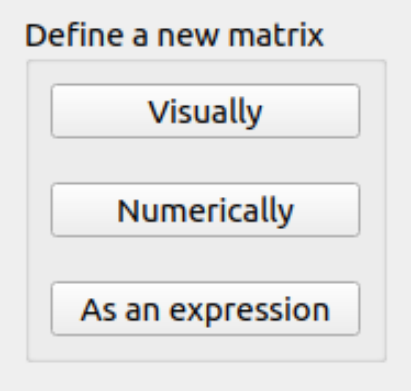


Figure 3.28: The new matrix definition buttons

3.8 Fumbling with SemVer

3.8.1 The first version numbers

At this point, I've been developing lintrans for quite a while, but I don't have any kind of version numbers yet. I wanted to fix this, to keep track of versions through history. SemVer[39] is a system of numbering versions of software that ensures compatibility across versions and gives semantic meaning to the version numbers.

My first foray into using SemVer was to declare `lintrans` at this point to be `0.1.0-alpha`. In retrospect, these early versions don't make much sense. I didn't release anything for anyone else to use until version `0.3.0`, so everything before then doesn't make much sense and should probably just be ignored.

I then made a few adjustments to syntax and documentation before declaring version `0.1.1-alpha` just over 24 hours later. This one was completely unnecessary in hindsight and I think I was just excited about being able to tag git commits. Additionally, there is a `__version__` variable in the root of the package, and the value of this variable is still `"0.1.0-alpha"` at the `v0.1.1-alpha` tag because I forgot to update it before I tagged the commit, and I didn't know how to fix it at the time.

3.8.2 Licensing

Using version numbers reminded me of licensing. I was already using the GNU GPLv3[14] for `lintrans` and most of my other projects, so I just wanted to add the copyright notice to all the source files of the project, as per the license. I already had the `COPYING` file in the project root with the full license, as required.

The notice looks like this[17]:

```
1  # lintrans - The linear transformation visualizer
2  # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4  # This program is licensed under GNU GPLv3, available here:
5  # <https://www.gnu.org/licenses/gpl-3.0.html>
```

3.8.3 Making the package executable

So far, I'd been running the program by running a separate script called `run_gui.py`, but Python lets you make a package executable by creating a file called `__main__.py` in the project root. You can then run it by using the command `python -m package_name`. This will make the whole program more cohesive by integrating the execution into the same directory as the source library code. I just moved the script here and changed its docstring a little bit.

```
   # 847cdf61f64b30e81edf0821b7a8d2eb060fb825
   # src/lintrans/__main__.py

9  """This module very simply runs the app by calling :func:`lintrans.gui.main_window.main`.
10
11 This allows the user to run the app like ``python -m lintrans`` from the command line.
12 """
13
14 import sys
15
16 from lintrans.gui import main_window
```

[17] I'm calling myself D. Dyson here because my name is Dyson, and I was in the process of getting it legally changed to Dyson Dyson when I made this change. I can't attribute this copyright to Dyson, since that's already a registered trademark of Dyson Limited. Calling myself Dyson Dyson felt strange, especially since it wasn't even my legal name at the time. So I chose to use D. Dyson instead.

```

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

Somehow this caused a conflict between my `lintrans.typing` package, which contained custom types like `MatrixType`, and the standard library `typing` package, so I had to rename mine to `lintrans.typing_`.

3.8.4 Improving the documentation

Throughout the whole project, I've been using GitHub Actions[13] to automatically do things like test and lint my code, as well as compile the documentation whenever I push my changes to GitHub. I recently learned about `pylint`'s ability to generate a graph of all the internal imports in a package[27]. I decided to generate this graph automatically and add it to my documentation.

```
# 39a3727fca69ea65571a15c55741578abce1e763
# .github/workflows/compile-docs.yaml

name: Compile docs for gh-pages

on:
  push:
    branches: [ main ]

jobs:
  compile-docs:
    runs-on: ubuntu-latest

    concurrency:
      group: ${{ github.workflow }}-${{ github.ref }}

    steps:
    - uses: actions/checkout@v2

    - name: Set up Python 3.10
      uses: actions/setup-python@v2
      with:
        python-version: '3.10'

    - name: Install dependencies
      run: |
        pip install --upgrade pip
        pip install -r requirements.txt -r docs/docs_requirements.txt
        pip install -e .
        pip install pylint
        sudo apt-get install -y graphviz

    - name: Create pylint import graphs
      run: |
        shopt -s globstar
        pylint --rcfile=/dev/null --exit-zero --reports=y --disable=all --enable=imports,RP0402
↪  --int-import-graph=docs/source/int-imports.png src/lintrans/**/*.py

    - name: Build docs
      run: cd docs/ && make html && cd ..

    - name: Deploy
      uses: peaceiris/actions-gh-pages@v3.8.0
      if: ${{ github.ref == 'refs/heads/main' }}
      with:
        github_token: ${{ secrets.GITHUB_TOKEN }}
        publish_dir: ./docs/build/html/
        keep_files: true
        destination_dir: docs
        user_name: 'github-actions[bot]'
        user_email: 'github-actions[bot]@users.noreply.github.com'
        commit_message: 'compile docs:'
```

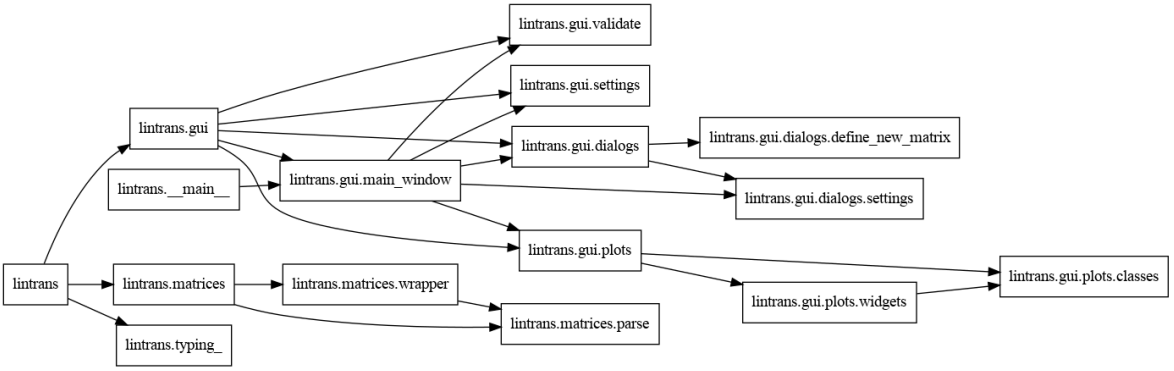


Figure 3.29: The internal imports graph

3.8.5 Fixing the colours

Now that I was using version numbers, I thought it was finally time to fix the colours in accordance with §2.4. I decided to use red and blue for the basis vectors, since these are the most easily distinguished colours for colour blind users. I decided to keep the green colour and use it for the eigenvectors and eigenlines. These are very visually distinct from the basis vectors, so I don't think I need to further distinguish them with colours. Red-green colour blind users should be able to tell them apart with ease.

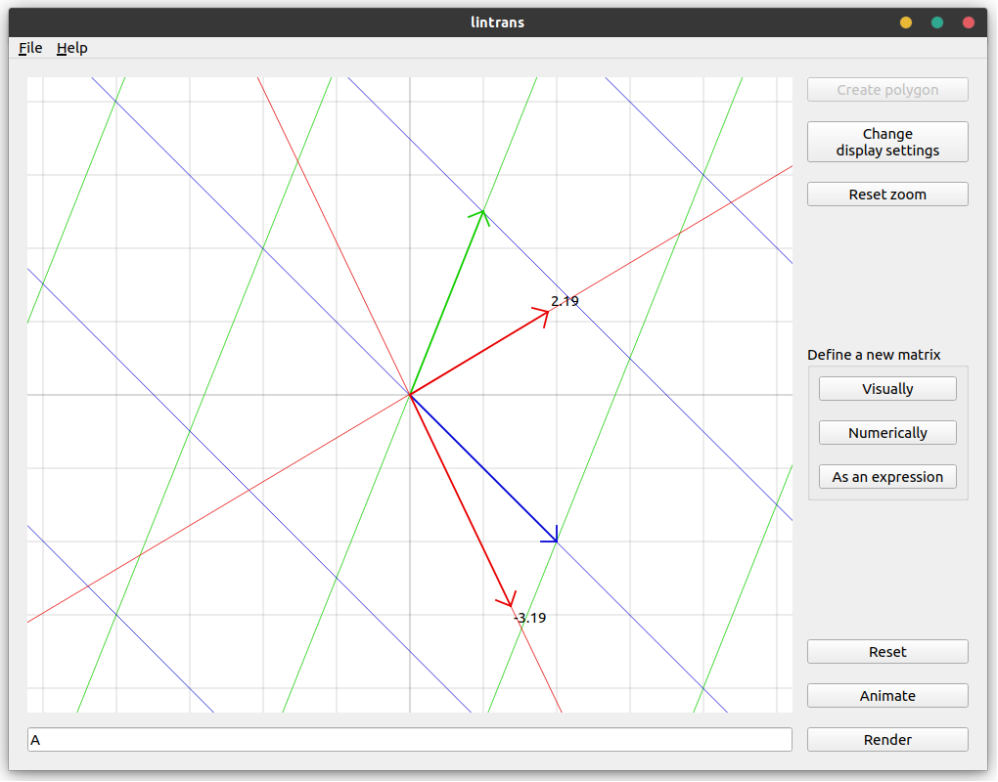


Figure 3.30: The old colours

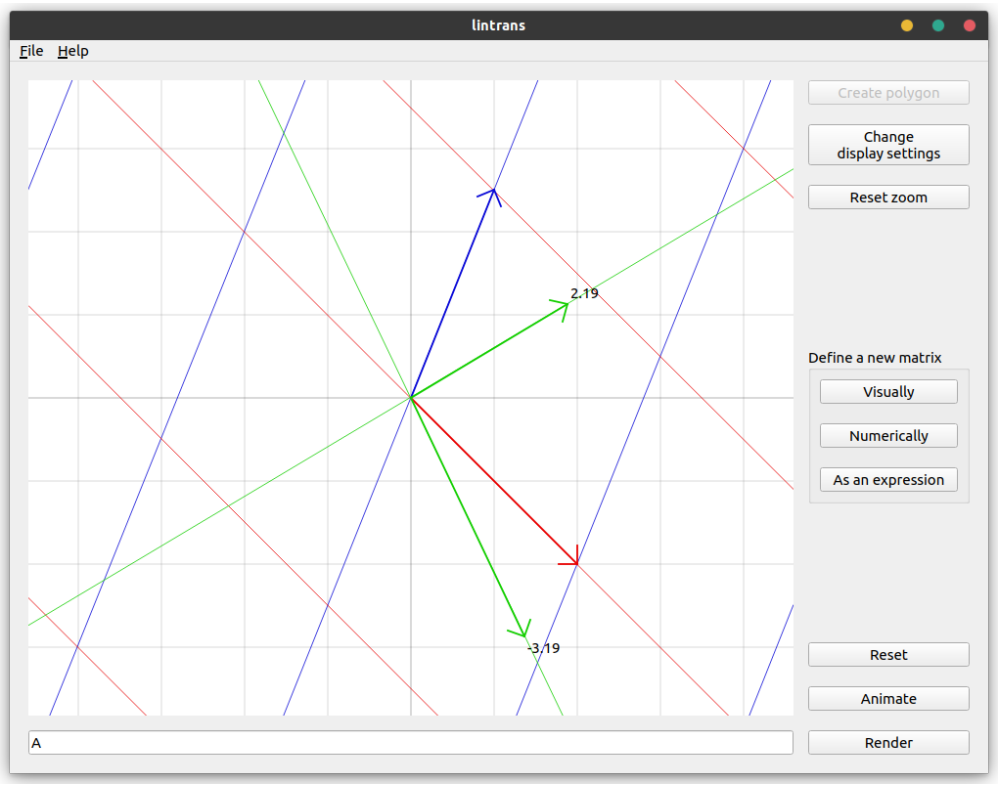


Figure 3.31: The new colours

3.8.6 Compiling for release

At this point, I felt ready to show `lintrans` to some teachers. Obviously I didn't expect them to have a modern version of Python to run the program, especially not on a school laptop, so I had to compile the program. I had a look at things like Nuitka[23], but I had a few issues with it and it took ages to compile. Eventually I chose PyInstaller[25], which I've used to compile Python programs before. It's fast and effective and works very well.

I use Ubuntu as my daily driver, so I don't really want to spin up a Windows virtual machine and following some exact steps every time I want to compile `lintrans` for release, so I automated the process with GitHub Actions.

```yaml
# 9d0cbe69c596c04a07809a288d72b5e624f485d7
# .github/workflows/compile-release.yaml

name: Compile and release

on:
  push:
    tags:
      - 'v*.*.*'

jobs:
  check:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2

      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v2
        with:
          python-version: '3.10'

```

```yaml
      - name: Install dependencies
        run: |
          pip install --upgrade pip
          pip install -r requirements.txt -r dev_requirements.txt
          pip install -e .

      - name: Check
        run: |
          mypy src/ tests/
          flake8 src/ tests/
          pycodestyle src/ tests/
          pydocstyle src/ tests/

  test:
    needs: check
    runs-on: ${{ matrix.os }}

    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]

    steps:
      - uses: actions/checkout@v2

      - name: Set up Python 3.10
        uses: actions/setup-python@v2
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: |
          pip install --upgrade pip
          pip install -r requirements.txt -r dev_requirements.txt
          pip install -e .

      - name: Test
        run: |
          pytest
          pytest --doctest-modules src/

  compile:
    needs: test
    runs-on: ${{ matrix.os }}

    strategy:
      matrix:
        os: [ubuntu-latest, macos-latest, windows-latest]

    steps:
      - uses: actions/checkout@v2

      - name: Set up Python 3.10
        uses: actions/setup-python@v2
        with:
          python-version: '3.10'

      - name: Install dependencies
        run: |
          pip install --upgrade pip
          pip install -r requirements.txt -r dev_requirements.txt
          pip install -e .
          pip install pyinstaller

      - name: Compile
        run: pyinstaller --onefile --windowed --distpath=./dist --name lintrans-${{ runner.os }}
↪  src/lintrans/__main__.py

      - name: Upload artifact
        uses: actions/upload-artifact@v3
        with:
          name: ${{ matrix.os }}-binary
          path: dist/lintrans*
```

```yaml
  publish:
    needs: compile
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2

      - name: Download Windows binary
        uses: actions/download-artifact@v3
        with:
          name: windows-latest-binary
          path: dist/

      - name: Download macOS binary
        uses: actions/download-artifact@v3
        with:
          name: macos-latest-binary
          path: dist/

      - name: Download Linux binary
        uses: actions/download-artifact@v3
        with:
          name: ubuntu-latest-binary
          path: dist/

      - name: Check for alpha
        id: checkalpha
        run: |
          isalpha=$(if [ -n "$(echo $GITHUB_REF | grep -o -- 'alpha')" ]; then echo 1; else echo 0; fi)
          echo "::set-output name=isalpha::$isalpha"

      - name: Upload binaries (normal release)
        if: steps.checkalpha.outputs.isalpha == 0
        uses: softprops/action-gh-release@v1
        with:
          fail_on_unmatched_files: true
          prerelease: false
          draft: true  # I'll manually add the title and description when the draft is online
          files: |
            dist/lintrans-Windows.exe
            dist/lintrans-macOS
            dist/lintrans-Linux

      - name: Upload binaries (pre-release)
        if: steps.checkalpha.outputs.isalpha == 1
        uses: softprops/action-gh-release@v1
        with:
          fail_on_unmatched_files: true
          prerelease: true
          draft: true  # I'll manually add the title and description when the draft is online
          files: |
            dist/lintrans-Windows.exe
            dist/lintrans-macOS
            dist/lintrans-Linux
```

This workflow only runs when I push a tag containing a version number. It firstly installs Python[18] and then lints and tests everything. If that all passes, then it compiles `lintrans` on Ubuntu, macOS, and Windows. Then it publishes the compiled binaries to a GitHub draft release. I then go in when it's finished to add a description to the release and publish it.

Now that I had this in place, I released version `v0.2.0-alpha`.

After some minor adjustments, such as including the tag name in the binary name, and installing UPX[46] for the Linux compilation[19], I released `v0.2.0` just 6 hours later.

[18] There's a mistake on line 14. There is no strategy matrix, so '`${{ matrix.python-version }}`' is treated as plain text rather than an actual version number. This was caused by copying a previous workflow. The actual version that it installs is fine, though. It's only the name of the step that doesn't look right.

[19] UPX is designed to compress executable files, but I later learned that PyInstaller doesn't even use UPX on Linux; it only uses it on Windows.

I made a few minor adjustments to the workflows, but soon moved on from this version number fiasco and started doing more actual work.

3.9 Preparing for `v0.2.1`

3.9.1 Fixing slots and signals

I was perusing the Qt5 documentation when I learned about the difference between the `dialog.finished` signal and the `dialog.accepted` signal. I decided to rework some old code to make better use these signals.

When defining a new matrix or dialog settings, we only want to save the new data if the user actually accepted the dialog by clicking the confirm button. We don't want to save it if they clicked cancel. However, in the case of the error message dialog, we always want to update the render buttons when it's closed, no matter how the user closed the dialog.

```
# 66242465222a153a5f37c4a1a3c2bd50bfd90933
# src/lintrans/gui/main_window.py

35  class LintransMainWindow(QMainWindow):
...
447         @pyqtSlot(DefineDialog)
448         def dialog_define_matrix(self, dialog_class: Type[DefineDialog]) -> None:
...
461             # We create a dialog with a deepcopy of the current matrix_wrapper
462             # This avoids the dialog mutating this one
463             dialog = dialog_class(deepcopy(self.matrix_wrapper), self)
464
465             # .open() is asynchronous and doesn't spawn a new event loop, but the dialog is still modal (blocking)
466             dialog.open()
467
468             # So we have to use the accepted signal to call a method when the user accepts the dialog
469             dialog.accepted.connect(self.assign_matrix_wrapper)
...
478         @pyqtSlot()
479         def dialog_change_display_settings(self) -> None:
480             """Open the dialog to change the display settings."""
481             dialog = DisplaySettingsDialog(self.plot.display_settings, self)
482             dialog.open()
483             dialog.accepted.connect(lambda: self.assign_display_settings(dialog.display_settings))
...
493         def show_error_message(self, title: str, text: str, info: str | None = None) -> None:
...
511             # This is `finished` rather than `accepted` because we want to update the buttons no matter what
512             dialog.finished.connect(self.update_render_buttons)
```

I also added the `@pyqtSlot()` decorator to all the relevant methods in the matrix definition dialogs. The types in the brackets indicate the signature of the method.

A slot in Qt5 is just a method that is expected to be connected to a signal, so it gets called from the event loop. Using the decorator makes it clear that a method is a slot, and also allows slightly better performance.

```
# 9beff9cf25d3af655e134205572a5668279f42cc
# src/lintrans/gui/dialogs/define_new_matrix.py

67  class DefineDialog(QDialog):
...
138         @abc.abstractmethod
139         @pyqtSlot()
140         def update_confirm_button(self) -> None:
...
143         @pyqtSlot(int)
144         def load_matrix(self, index: int) -> None:
...
155         @abc.abstractmethod
156         @pyqtSlot()
157         def confirm_matrix(self) -> None:
```

```
...
164     class DefineVisuallyDialog(DefineDialog):
...
194         @pyqtSlot()
195         def update_confirm_button(self) -> None:
...
203         @pyqtSlot(int)
204         def load_matrix(self, index: int) -> None:
...
214         @pyqtSlot()
215         def confirm_matrix(self) -> None:
...
226     class DefineNumericallyDialog(DefineDialog):
...
276         @pyqtSlot()
277         def update_confirm_button(self) -> None:
...
288         @pyqtSlot(int)
289         def load_matrix(self, index: int) -> None:
...
305         @pyqtSlot()
306         def confirm_matrix(self) -> None:
...
317     class DefineAsAnExpressionDialog(DefineDialog):
...
348         @pyqtSlot()
349         def update_confirm_button(self) -> None:
...
356         @pyqtSlot(int)
357         def load_matrix(self, index: int) -> None:
...
364         @pyqtSlot()
365         def confirm_matrix(self) -> None:
```

3.9.2 Linking in documentation

I've been using Sphinx[42] for my documentation this whole time, and I've been using the Sphinx extension `intersphinx` to link to the Python standard library documentation. It uses a system of binary inventory files which define a reference map between names to use in the documentation, and where to link those names to. I recently learned of the `sphobjinv` Python package, which allows you to easily create your own local inventory files to reference some external source of documentation, such as the Qt5 documentation. I read through the `sphobjinv` documentation[41] and designed a small script to read a custom text file, and create the binary inventory file needed by Sphinx.

```
# 5455265a51666e29ab976152c1a758a422e1004a
# docs/create_objects_inv.py

 9  """A simple script to convert my manually curated text file to an invetory file that intersphinx can use.
10
11  .. note:: The URIs in the text file must not have .html suffices
12  """
13
14  import re
15  from glob import glob
16
17  import sphobjinv as soi
18
19
20  pattern = re.compile(r'^(\S+)\s+([^:\s]+):([^:\s]+)\s+(\d+)\s+(\S+)\s+(\S+)$')
21
22
23  def generate_objects_inv(prefix: str) -> None:
24      """Generate the ``objects.inv`` file for PyQt5.
25
26      We read from ``prefix-objects.txt`` and write to ``prefix-objects.inv``,
27      so if you want to use ``pyqt5-objects.txt``, then the prefix should be ``pyqt5``.
28
29      :param str prefix: The prefix for the object files
```

```
30          """
31          inv = soi.Inventory()
32          inv.project = 'PyQt5'
33          inv.version = '5.15'
34
35          with open(prefix + '-objects.txt', 'r', encoding='utf-8') as f:
36              text = f.read().splitlines()
37
38          for line in text:
39              if line == '' or line.lstrip().startswith('#'):
40                  continue
41
42              if (match := re.match(pattern, line)) is None:
43                  raise ValueError(f'Every line in {prefix}-objects.txt must match the pattern')
44
45              name, domain, role, priority, uri, disp_name = match.groups()
46
47              inv.objects.append(soi.DataObjStr(
48                  name=name, domain=domain, role=role, priority=priority, uri=uri, dispname=disp_name
49              ))
50
51          compressed_text = soi.compress(inv.data_file(contract=True))
52          soi.writebytes(f'source/{prefix}-objects.inv', compressed_text)
53
54
55      def main() -> None:
56          """Call :func:`generate_objects_inv` for every file matching the glob pattern '*-objects.txt'."""
57          for filename in glob('*-objects.txt'):
58              prefix = filename[:-12]
59              print(f'Generating {prefix}-objects.inv')
60              generate_objects_inv(prefix)
61
62
63      if __name__ == '__main__':
64          main()
```

Line 11 should say ‘must have .html suffices’.

```
# aab8e88b0e2cdae8038c9935031c74bcdae0ad5c
# docs/pyqt5-objects.txt

1   # This format is:
2   # <reference name> <domain>:<role> <priority> <URI> <display name>
3   # Sphinx handles the prefix for the URI
4   # If the display name is '-', then it's the same as the reference name
5
6   # === Classes
7
8   QApplication py:class 1 qapplication.html -
9   QDialog      py:class 1 qdialog.html      -
10  QKeyEvent    py:class 1 qkeyevent.html    -
11  QPainter     py:class 1 qpainter.html     -
12  QPaintEvent  py:class 1 qpaintevent.html  -
13  QWheelEvent  py:class 1 qwheelevent.html  -
14  QWidget      py:class 1 qwidget.html      -
15
16  # === Methods
17
18  QPainter.drawLine:iiii py:method 1 qpainter.html#drawLine-2 QPainter.drawLine()
19
20  # === Signals
21
22  QComboBox.activated py:method 1 qcombobox.html#activated -
23  QDialog.accepted    py:method 1 qdialog.html#accepted    -
24
25  # These are in full form so that autodoc can reference base classes and param types
26
27  PyQt5.QtGui.QKeyEvent   py:class 1 qkeyevent.html   -
28  PyQt5.QtGui.QPainter    py:class 1 qpainter.html    -
29  PyQt5.QtGui.QPaintEvent py:class 1 qpaintevent.html -
30  PyQt5.QtGui.QWheelEvent py:class 1 qwheelevent.html -
```

```
31      PyQt5.QtWidgets.QDialog py:class 1 qdialog.html     -
32      PyQt5.QtWidgets.QWidget py:class 1 qwidget.html     -
```

I then just had to change all the references to Qt5 things in the documentation and then Sphinx would automatically link all the Qt5 references to their appropriate links, as defined in this file.

3.9.3 Improving tests

I made some small improvements to the unit tests by making sure they handled greedy index parsing, which means that something like `A^2 3B` will get parsed as `A^{23}B` because whitespace is ignored, as well asserting that all invalid expressions raise `MatrixParseError`. I also added the copyright comment to the test files.

This was a tiny change, but worth noting.

```
        # c07d97024e1fe00ab110f43e5c7e6737c955d680
        # tests/matrices/test_parse_and_validate_expression.py

41      expressions_and_parsed_expressions: list[tuple[str, MatrixParseList]] = [
...
50          ('A^12', [[('', 'A', '12')]]),
51          ('A^234', [[('', 'A', '234')]]),
...
55          ('A^2 3B', [[('', 'A', '23'), ('', 'B', '')]]),
...
72          ('2.14A^{3} 4.5rot(14.5)^-1 + 8.5B^T 5.97C^14 - 3.14D^{-1} 6.7E^T',
73           [[('2.14', 'A', '3'), ('4.5', 'rot(14.5)', '-1')], [('8.5', 'B', 'T'), ('5.97', 'C', '14')],
...
75      ]
...
78      def test_parse_matrix_expression() -> None:
79          """Test the parse_matrix_expression() function."""
80          for expression, parsed_expression in expressions_and_parsed_expressions:
81              # Test it with and without whitespace
82              assert parse_matrix_expression(expression) == parsed_expression
83              assert parse_matrix_expression(expression.replace(' ', '')) == parsed_expression


        # 70e1a7271a61f3009cc4d342f46743b248498a1c
        # tests/matrices/test_parse_and_validate_expression.py

26      invalid_inputs: list[str] = [
27          '', 'rot()', 'A^', 'A^1.2', 'A^{3.4}', '1,2A', 'ro(12)', '5', '12^2', '^T', '^{12}',
28          'A^{13', 'A^3}', 'A^A', '^2', 'A--B', '--A', '+A', '--1A', 'A--B', 'A--1B', '.A', '1.A'
29
30          'This is 100% a valid matrix expression, I swear'
31      ]
...
86      def test_parse_error() -> None:
87          """Test that parse_matrix_expression() raises a MatrixParseError."""
88          for expression in invalid_inputs:
89              with pytest.raises(MatrixParseError):
90                  parse_matrix_expression(expression)
```

3.9.4 The Windows version file

Windows stores metadata for `.exe` files inside the `.exe` files themselve[47][48]. PyInstaller lets you embed that metadata when compiling on Windows, using a version file[26]. Obviously, I wanted to include this metadata in my compiled `.exe` in the release.

I created a simple `precompile.py` script that would create different pre-compilation artefacts. I could then use these in the compilation workflow for GitHub Actions. I started with just Windows, but obviously I'm going to expand into Linux and macOS as well later.

```
# 2126959cb6f836b1bc6c92dad859b43cbd86e1ab
# precompile.py

"""A simple pre-compile script for the automated GitHub page compilation action."""

import re
import sys


def precompile_macos() -> None:
    """Run the pre-compile steps for macOS."""
    print('Pre-compile for macOS not implemented yet')


def precompile_windows(args: list[str]) -> None:
    """Run the pre-compile steps for Windows."""
    print('Pre-compiling for Windows')

    if len(args) < 1:
        raise ValueError('Windows pre-compile needs tag name argument')

    tag_name = args[0]

    if (m := re.match(r'v(\d+)\.(\d+)\.(\d+)(-alpha)?', tag_name)) is not None:
        major, minor, patch, alpha = m.groups()

    else:
        raise ValueError('Tag name must match format')

    if alpha is not None:
        flags = '0x2'
    else:
        flags = '0x0'

    version_tuple = f'{major}, {minor}, {patch}, 0'

    version_info = f'''VSVersionInfo(
  ffi=FixedFileInfo(
    filevers=({version_tuple}),
    prodvers=({version_tuple}),
    mask=0x3f,
    flags={flags},
    OS=0x40004,
    fileType=0x1,
    subtype=0x0,
    date=(0, 0)
  ),
  kids=[
    StringFileInfo(
      [
        StringTable(
          '040904B0',
          kids=[
            StringStruct('CompanyName', 'D. Dyson (DoctorDalek1963)'),
            StringStruct('FileDescription', 'Linear transformation visualizer'),
            StringStruct('FileVersion', '{tag_name}'),
            StringStruct('InternalName', 'lintrans'),
            StringStruct('LegalCopyright', '(C) D. Dyson (DoctorDalek1963) under GPLv3'),
            StringStruct('OriginalFilename', 'lintrans-Windows-{tag_name}.exe'),
            StringStruct('ProductName', 'lintrans'),
            StringStruct('ProductVersion', '{tag_name}')
          ]
        )
      ]
    ),
    VarFileInfo([VarStruct('Translation', [2057, 1200])])
  ]
)
'''

    with open('version_info.txt', 'w', encoding='utf-8') as f:
        f.write(version_info)

```

```
79          print('Version file written to version_info.txt')
80
81
82      def main(args: list[str]) -> None:
83          """Evaluate the arguments and pre-compile accordingly."""
84          if len(args) < 1:
85              raise ValueError('Script must be supplied with the name of an operating system.')
86
87          os_name = args[0].lower()
88
89          if os_name == 'linux':
90              print("Linux doesn't need any pre-compilation")
91
92          elif os_name == 'macos':
93              precompile_macos()
94
95          elif os_name == 'windows':
96              precompile_windows(args[1:])
97
98          else:
99              raise ValueError(f'Unsupported operating system "{os_name}"')
100
101
102     if __name__ == '__main__':
103         main(sys.argv[1:])
```

```
# 2126959cb6f836b1bc6c92dad859b43cbd86e1ab
# .github/workflows/compile-release.yaml

  8  jobs:
...
 60    compile:
...
 68      steps:
...
 87        - name: Pre-compile
 88          run: python precompile.py ${{ runner.os }} $GITHUB_REF_NAME
 89          shell: bash
 90
 91        - name: Compile (Linux)
 92          if: runner.os == 'Linux'
 93          run: pyinstaller --onefile --windowed --distpath=./dist --name lintrans-${{ runner.os }}-$GITHUB_REF_NAME
↪  src/lintrans/__main__.py
 94          shell: bash
 95
 96        - name: Compile (macOS)
 97          if: runner.os == 'macOS'
 98          run: pyinstaller --onefile --windowed --distpath=./dist --name lintrans-${{ runner.os }}-$GITHUB_REF_NAME
↪  src/lintrans/__main__.py
 99          shell: bash
100
101       - name: Compile (Windows)
102         if: runner.os == 'Windows'
103         run: pyinstaller --onefile --windowed --distpath=./dist --version-file version_info.txt --name
↪  lintrans-${{ runner.os }}-$GITHUB_REF_NAME src/lintrans/__main__.py
104         shell: bash
```

I quickly realised that this design would make the compilation process more complicated, since the process would be in separate parts. Instead, I decided to create a unified compilation script, which runs a pre-compile step dependent on the operating system, and then compiles the program with the `PyInstaller.__main__.run()` function.

```
# ca674c7f7d61e8eed3410d456787cfe5b2bc28e5
# compile.py

 9  """A simple pre-compile script for the automated GitHub page compilation action."""
10
11  import argparse
12  import os
```

```
    import re
    import shutil
    import sys
    from textwrap import dedent

    from PyInstaller.__main__ import run as run_pyi

    import lintrans


    class Compiler:
        """A simple class to encapsulate compilation logic."""

        def __init__(
                self, *,
                platform: str | None = None,
                version_name: str | None = None,
                filename: str | None = None
        ):
            """Create a Compiler object."""
            self.platform = platform if platform else sys.platform
            self.version_name = version_name if version_name else lintrans.__version__
            self.filename = filename if filename else 'lintrans'

        def _precompile_windows(self) -> None:
            """Pre-compile for Windows."""
            if (m := re.match(r'v?(\d+)\.(\d+)\.(\d+)(-alpha)?', self.version_name)) is not None:
                major, minor, patch, alpha = m.groups()

            else:
                raise ValueError('Tag name must match format')

            if alpha is not None:
                flags = '0x2'
            else:
                flags = '0x0'

            version_tuple = f'{major}, {minor}, {patch}, 0'

            version_info = dedent(f'''
            VSVersionInfo(
              ffi=FixedFileInfo(
                filevers=({version_tuple}),
                prodvers=({version_tuple}),
                mask=0x3f,
                flags={flags},
                OS=0x40004,
                fileType=0x1,
                subtype=0x0,
                date=(0, 0)
              ),
              kids=[
                StringFileInfo(
                  [
                    StringTable(
                      '040904B0',
                      kids=[
                        StringStruct('CompanyName', 'D. Dyson (DoctorDalek1963)'),
                        StringStruct('FileDescription', 'Linear transformation visualizer'),
                        StringStruct('FileVersion', '{self.version_name}'),
                        StringStruct('InternalName', 'lintrans'),
                        StringStruct('LegalCopyright', '(C) D. Dyson (DoctorDalek1963) under GPLv3'),
                        StringStruct('OriginalFilename', '{self.filename}.exe'),
                        StringStruct('ProductName', 'lintrans'),
                        StringStruct('ProductVersion', '{self.version_name}')
                      ]
                    )
                  ]
                ),
                VarFileInfo([VarStruct('Translation', [2057, 1200])])
              ]
            )
            ''')
```

```
86
87              with open('version_info.txt', 'w', encoding='utf-8') as f:
88                  f.write(version_info)
89
90              print('Version file written to version_info.txt')
91
92          def precompile(self) -> None:
93              """Pre-compile for the appropriate operating system."""
94              if self.platform == 'linux':
95                  print("Linux doesn't need any pre-compilation")
96
97              elif self.platform == 'darwin':
98                  print("macOS doesn't need any pre-compilation")
99
100             elif self.platform == 'win32':
101                 print('Pre-compiling for Windows')
102                 self._precompile_windows()
103
104             else:
105                 raise ValueError(f'Unsupported operating system "{self.platform}"')
106
107         def _get_pyi_args(self) -> list[str]:
108             """Return the common args for PyInstaller."""
109             return [
110                 'src/lintrans/__main__.py',
111                 '--onefile',
112                 '--windowed',
113                 '--distpath=./dist',
114                 '--workpath=./build',
115                 '--noconfirm',
116                 '--clean',
117                 f'--name={self.filename}'
118             ]
119
120         def _compile_macos(self) -> None:
121             """Compile for macOS."""
122             run_pyi(self._get_pyi_args())
123
124             os.rename(os.path.join('dist', self.filename + '.app'), self.filename + '.app')
125
126         def _compile_linux(self) -> None:
127             """Compile for Linux."""
128             run_pyi(self._get_pyi_args())
129
130             os.rename(os.path.join('dist', self.filename), self.filename)
131
132         def _compile_windows(self) -> None:
133             """Compile for Windows."""
134             if not os.path.isfile('version_info.txt'):
135                 raise ValueError('Windows compilation requires version_info.txt from pre-compilation')
136
137             run_pyi([
138                 *self._get_pyi_args(),
139                 '--version-file',
140                 'version_info.txt'
141             ])
142
143             os.remove('version_info.txt')
144
145             os.rename(os.path.join('dist', self.filename + '.exe'), self.filename + '.exe')
146
147         def compile(self) -> None:
148             """Compile for the appropriate operating system."""
149             if self.platform == 'darwin':
150                 self._compile_macos()
151
152             elif self.platform == 'linux':
153                 self._compile_linux()
154
155             elif self.platform == 'win32':
156                 self._compile_windows()
157
158             else:
```

```
159                 raise ValueError(f'Unsupported operating system "{self.platform}"')
160
161             shutil.rmtree('dist')
162             shutil.rmtree('build')
163             os.remove(self.filename + '.spec')
164
165
166     def main() -> None:
167         """Run any pre-compilation, and then compile."""
168         parser = argparse.ArgumentParser(description='Compile this version of lintrans for your operating system',
            ↪  add_help=True)
169         parser.add_argument('-f', '--filename', type=str, required=False, default=None, help='the filename (without
            ↪  extension)')
170         parser.add_argument('-v', '--version', type=str, required=False, default=None, help='the version name in the
            ↪  format v1.2.3')
171
172         args = parser.parse_args()
173
174         compiler = Compiler(filename=args.filename, version_name=args.version)
175         compiler.precompile()
176         compiler.compile()
177
178
179     if __name__ == '__main__':
180         main()
```

This new compilation script captures the whole process. On Linux or macOS, it just compiles the program with PyInstaller. On Windows, it has to generate the version file, then compile the program with an additional argument to include the version file. I then updated the GitHub Actions workflow to use this new compilation script.

```
# e47fe732954bf018128bdcb5ee9c354910517f36
# .github/workflows/compile-release.yaml

 8  jobs:
...
60    compile:
...
68      steps:
...
87        - name: Compile
88          run: python compile.py -f lintrans-${{ runner.os }}-${{ env.GITHUB_REF_NAME }} -v $GITHUB_REF_NAME
```

3.9.5 Compiling for macOS

Compiling for macOS is considerably more difficult. I run Ubuntu as my primary operating system. I used to use Windows and I have a Windows 10 virtual machine, so compiling for Windows was never an issue. But I've never used a Mac, and Apple don't like people installing macOS on non-Apple hardware. Getting a virtual machine set up running macOS Monterey was quite difficult, but I eventually managed to get it working somewhat properly.

Once I had `lintrans` compiling on macOS, I wanted to do something similar to what I did with Windows, where I added extra metadata to the executable. In macOS, executables are bundled as `.app` files, which are just directory structures, containing the necessary files and metadata[11]. All the metadata for an application bundle is contained in the `Info.plist` file in the bundle[9][8]. After reading the documentation for these files, I created one for `lintrans`.

```
# e716000521c92259ed4a1b33ab37e3860a7b7875
# compile.py

23  class Compiler:
...
92      def _macos_replace_info_plist(self) -> None:
93          """Replace the Info.plist file in the macOS app."""
```

```
94              short_version_name = self.version_name
95
96              if short_version_name.startswith('v'):
97                  short_version_name = short_version_name[1:]
98
99              if short_version_name.endswith('-alpha'):
100                 short_version_name = short_version_name[:-6]
101
102             new_info_plist = dedent(f'''
103             <?xml version="1.0" encoding="UTF-8"?>
104             <!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyList-1.0.dtd"
105             <plist version="1.0">
106             <dict>
107                 <key>CFBundleDisplayName</key>
108                 <string>lintrans</string>
109                 <key>CFBundleExecutable</key>
110                 <string>lintrans</string>
111                 <key>CFBundleIconFile</key>
112                 <string>icon-windowed.icns</string>
113                 <key>CFBundleIdentifier</key>
114                 <string>lintrans</string>
115                 <key>CFBundleInfoDictionaryVersion</key>
116                 <string>6.0</string>
117                 <key>CFBundleName</key>
118                 <string>lintrans</string>
119                 <key>CFBundleType</key>
120                 <string>APPL</string>
121                 <key>CFBundleVersion</key>
122                 <string>{self.version_name}</string>
123                 <key>CFBundleShortVersionString</key>
124                 <string>{short_version_name}</string>
125                 <key>NSHighResolutionCapable</key>
126                 <true/>
127                 <key>NSHumanReadableCopyright</key>
128                 <string>(C) D. Dyson (DoctorDalek1963) under GPLv3</string>
129             </dict>
130             </plist>
131             '''[1:])
132
133             with open(os.path.join(self.filename + '.app','Contents', 'Info.plist'), 'w', encoding='utf-8') as f:
134                 f.write(new_info_plist)
...
149         def _compile_macos(self) -> None:
150             """Compile for macOS."""
151             run_pyi(self._get_pyi_args())
152
153             os.rename(os.path.join('dist', self.filename + '.app'), self.filename + '.app')
154
155             self._macos_replace_info_plist()
```

This could would automatically generate the `Info.plist` file and write it to the correct place. However, I couldn't distribute this bundled app, since Apple requires an app to be signed by a trusted author before other people can download and run that code on their own Apple devices[7]. And unfortunately, Apple charges $99 per year for membership to their 'Apple Developer Program', which is required to become a trusted developer[10][20].

Microsoft also wants its developers to sign their code for security and trust, and also charges for this privilege. But on Windows, you can run a foreign, unsigned app so long as you dismiss the potential virus warning. On macOS, it's completely forbidden.

I don't exactly want to pay $99 per year to compile `lintrans` for macOS, when my audience on that platform is realistically almost zero, so I just removed macOS from the GitHub Actions compilation workflow. For anyone that wants to run `lintrans` on macOS, I will provide instructions to compile it from source.

[20] The price is listed in the small print at the bottom of the page.

3.9.6 Supporting flags

Most Linux apps support command line arguments and/or flags[21], and so I wanted to support these with `lintrans`. I wanted one flag to display help for the program, and one flag to display the version number of the program. Implementing this was very simple and I didn't bother using a library like `argparse`; I just used `sys.argv`.

```
# ffc603f1bf049811cb927f879ce7989456f6a537
# src/lintrans/__main__.py

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

import sys
from textwrap import dedent

from lintrans import __version__
from lintrans.gui import main_window


def main(prog_name: str, 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`.

    ``prog_name`` is ``sys.argv[0]`` when this script is run with ``python -m lintrans``.

    :param str prog_name: The name of the program
    :param list[str] args: The other arguments to the program
    """
    if '-h' in args or '--help' in args:
        print(dedent(f'''
        Usage: {prog_name} [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:]))

    elif '-V' in args or '--version' in args:
        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:]))

    else:
        main_window.main(args)


if __name__ == '__main__':
    main(sys.argv[0], sys.argv[1:])
```

Here is the expected output when using these flags at a shell prompt:

```
$ python -m lintrans --help
Usage: /home/dyson/repos/lintrans/src/lintrans/__main__.py [option]

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

[21] For explanation and examples, see `https://betterdev.blog/command-line-arguments-anatomy-explained/`

```
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.
$ python -m lintrans --version
lintrans (version 0.2.0)
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>
```

3.9.7 Adding the about dialog

In addition to the help and version information available from the command line, most apps with a GUI provide a graphical way to access information about the app, typically in the form of an about box. I wanted to do this for `lintrans`. I had intended to have this from the start, but it now felt like the right time to implement it, since I was preparing to release version `0.2.1`.

To do this, I create a new file to house the new dialog class, and then added the functionality to the original button, so that it would now open the new dialog.

```
# 7423fff72f09b5f5a3253c734d42c4e7c3182efe
# src/lintrans/gui/dialogs/misc.py

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

from __future__ import annotations

import platform

from PyQt5 import QtWidgets
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QDialog, QVBoxLayout

import lintrans


class AboutDialog(QDialog):
    """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.
    """

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

```

```
49              label_info = QtWidgets.QLabel(self)
50              label_info.setText(
51                  'lintrans is a program designed to help visualise<br>'
52                  '2D linear transformations represented with matrices.<br><br>'
53                  "It's designed for teachers and students and any feedback<br>"
54                  'is greatly appreciated at <a href="https://github.com/DoctorDalek1963/lintrans" '
55                  'style="color: black;">my GitHub page</a><br>or via email '
56                  '(<a href="mailto:dyson.dyson@icloud.com" style="color: black;">dyson.dyson@icloud.com</a>).'
57              )
58              label_info.setAlignment(Qt.AlignCenter)
59              label_info.setTextFormat(Qt.RichText)
60              label_info.setOpenExternalLinks(True)
61
62              label_copyright = QtWidgets.QLabel(self)
63              label_copyright.setText(
64                  'This program is free software.<br>Copyright 2021-2022 D. Dyson (DoctorDalek1963).<br>'
65                  'This program is licensed under GPLv3, which can be found '
66                  '<a href="https://www.gnu.org/licenses/gpl-3.0.html" style="color: black;">here</a>.'
67              )
68              label_copyright.setAlignment(Qt.AlignCenter)
69              label_copyright.setTextFormat(Qt.RichText)
70              label_copyright.setOpenExternalLinks(True)
71
72              # === Arrange the widgets
73
74              self.setContentsMargins(10, 10, 10, 10)
75
76              vlay = QVBoxLayout()
77              vlay.setSpacing(20)
78              vlay.addWidget(label_title)
79              vlay.addWidget(label_version_info)
80              vlay.addWidget(label_info)
81              vlay.addWidget(label_copyright)
82
83              self.setLayout(vlay)
84
85              self.setFixedSize(self.baseSize())
```

```
# 7423fff72f09b5f5a3253c734d42c4e7c3182efe
# src/lintrans/gui/main_window.py

36  class LintransMainWindow(QMainWindow):
...
42      def __init__(self):
...
64          self.menu_help = QtWidgets.QMenu(self.menubar)
65          self.menu_help.setTitle('&Help')
...
97          self.action_about = QtWidgets.QAction(self)
98          self.action_about.setText('&About')
99          self.action_about.triggered.connect(lambda: dialogs.AboutDialog(self).open())
...
117         self.menu_help.addAction(self.action_about)
```

This feature works just as expected.

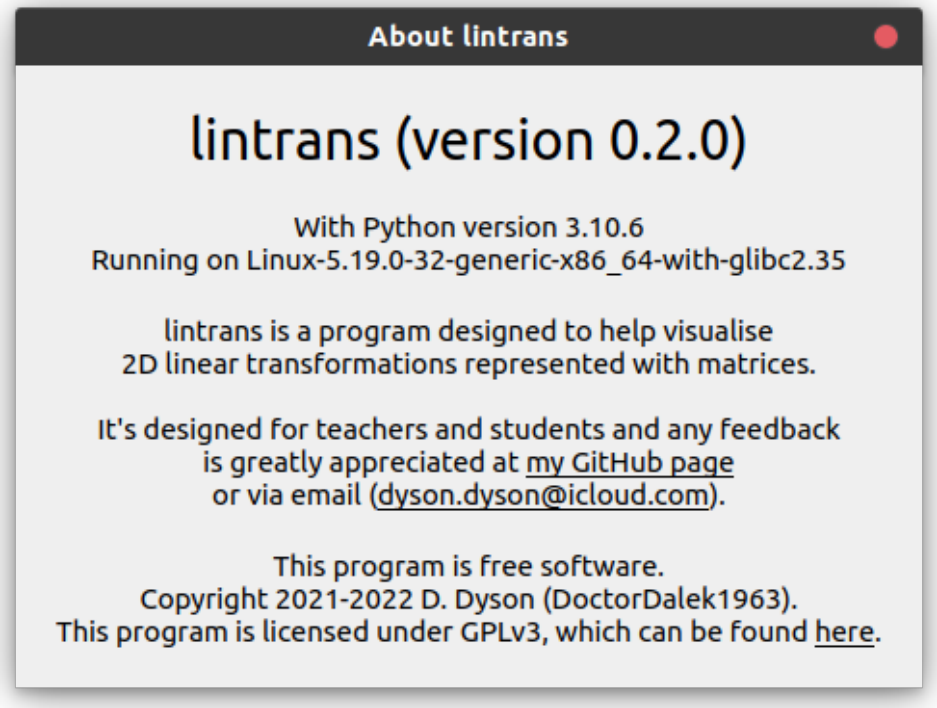


Figure 3.32: The about dialog

3.9.8 Creating the `FixedSizeDialog` class

Currently, several dialogs are a fixed size, meaning they can't be resized. Qt5 doesn't have the option to make a dialog non-resizeable - that's only available for whole windows - so to achieve this effect, I had to add the line `self.setFixedSize(self.baseSize())` to the end of the constructor for every fixed size dialog[22]. This was easy to forget and it didn't make it immediately clear which dialogs were fixed size and which weren't.

To fix this, I decided to use a different technique, by overriding the `open()` method of the dialog. This method gets called when the dialog is first opened. If I called the superclass' implementation first to do all the actual opening logic, then I could use that same previous technique to set the fixed size to whatever Qt5 had calculated it to be. By overriding a method, I could put that override into a simple superclass called `FixedSizeDialog` and then make all fixed size dialogs inherit from this class instead of just `QDialog`.

```
# a59ea87fb37fc593cf0bae3066bdbd24be656798
# src/lintrans/gui/dialogs/misc.py

20  class FixedSizeDialog(QDialog):
21      """A simple superclass to create modal dialog boxes with fixed size.
22
23      We override the :meth:`open` method to set the fixed size as soon as the dialog is opened modally.
24      """
25
26      def open(self) -> None:
27          """Override :meth:`QDialog.open` to set the dialog to a fixed size."""
28          super().open()
29          self.setFixedSize(self.size())
...
32  class AboutDialog(FixedSizeDialog):


# a59ea87fb37fc593cf0bae3066bdbd24be656798
```

[22] This works because Qt5 needs to arrange all the widgets and allocate space for all of them first, and then we can use that calculated size as the fixed size.

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

68  class DefineDialog(FixedSizeDialog):


# a59ea87fb37fc593cf0bae3066bdbd24be656798
# src/lintrans/gui/dialogs/settings.py

21  class SettingsDialog(FixedSizeDialog):
```

I used `self.size()` here instead of `self.baseSize()` like before. There doesn't seem to be much difference, but the docs seem to suggest that `self.size()` is more appropriate for this use case[34][37].

3.9.9 Increasing minimum grid spacing

When you zoom out, the grid lines get closer and closer together. I previously limited this to 1 pixel, since the program would hang or crash if it was any lower. However, having the grid lines that close together made it very hard to see what was happening. I could easily fix this by limiting the minimum grid spacing to say, 5 pixels.

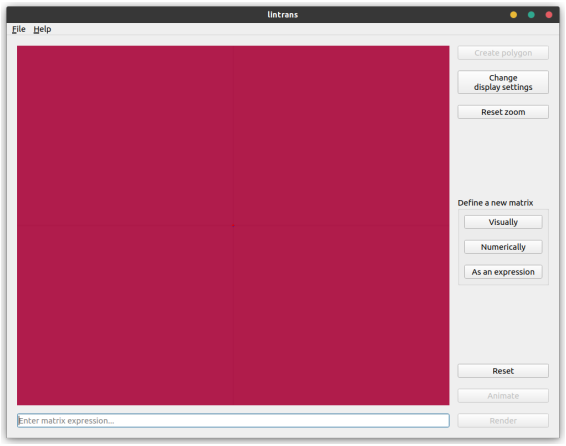


Figure 3.33: Fully zoomed out before

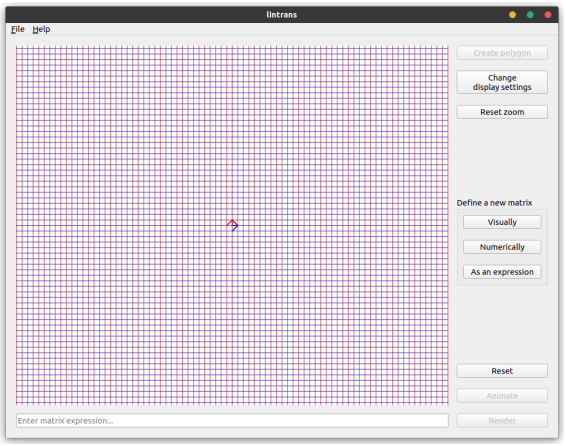


Figure 3.34: Fully zoomed out after

```
# 51d35018c81deaaf7e32646b1f99d6e46643bf24
# src/lintrans/gui/plots/classes.py

23  class BackgroundPlot(QWidget):
...
39      minimum_grid_spacing: int = 5
...
154     def wheelEvent(self, event: QWheelEvent) -> None:
155         """Handle a :class:`QWheelEvent` by zooming in or our of the grid."""
156         # angleDelta() returns a number of units equal to 8 times the number of degrees rotated
157         degrees = event.angleDelta() / 8
158
159         if degrees is not None:
160             new_spacing = max(1, self.grid_spacing + degrees.y())
161
162             if new_spacing >= self.minimum_grid_spacing:
163                 self.grid_spacing = new_spacing
164
165         event.accept()
166         self.update()
```

3.9.10 Creating a changelog

I've know about changelogs for a long time. They're used to keep track of changes to a project over time. I recently learned about the project 'keep a changelog', which tries to create a standard for changelog formats[19]. I want to have a proper changelog for lintrans, so I decided to implement one.

```
<!-- 47c68c7f4780d0e2c374cf12b9b54c031277af6d -->
<!-- CHANGELOG.md -->

# Changelog

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](https://keepachangelog.com/en/1.0.0/),
and this project adheres to [Semantic Versioning](https://semver.org/spec/v2.0.0.html).

## [Unreleased]

### Added

- Explicit `@pyqtSlot` decorators
- Link to Qt5 docs in project docs with intersphinx
- Copyright comment in tests and `setup.py`
- Create version file for Windows compilation
- Create full compile.py script
- Add `Info.plist` file for macOS compilation
- Support --help and --version flags in `__main__.py`
- Create about dialog in help menu
- Implement minimum grid spacing

### Fixed

- Fix problems with compile script
- Fix small bugs and docstrings

## [0.2.0] - 2022-03-11

There were alpha tags before this, but I wasn't properly adhering to semantic versioning, so I'll start the changelog here.

If I'd been using semantic versioning from the start, there would much more changelog here, but instead, I'll just summarise the features.

### Added

- Matrix context with the `MatrixWrapper` class
- Parsing and evaluating matrix expressions
- A simple GUI with a viewport to render linear transformations
- Simple dialogs to create matrices and assign them to names
- Ability to render and animate linear transformations parsed from defined matrices
- Ability to zoom in and out of the viewport
- Add dialog to change display settings

[Unreleased]: https://github.com/DoctorDalek1963/lintrans/compare/v0.2.0...HEAD
[0.2.0]: https://github.com/DoctorDalek1963/lintrans/compare/13600cc6ff6299dc4a8101a367bc52fe08607554...v0.2.0
```

3.9.11 Releasing `v0.2.1`

Now all I had to do for the release was update the version number in `__init__.py` and update the changelog to include the changes under the correct heading, and add a new, empty 'Unreleased' heading.

```
# d47f63eb0bcdc89cff5ed19afe2fc63899edf1d9
# src/lintrans/__init__.py

__version__ = '0.2.1'
```

```
<!-- d47f63eb0bcdc89cff5ed19afe2fc63899edf1d9 -->
<!-- CHANGELOG.md -->
```

```
# Changelog

All notable changes to this project will be documented in this file.

The format is based on [Keep a Changelog](https://keepachangelog.com/en/1.0.0/),
and this project adheres to [Semantic Versioning](https://semver.org/spec/v2.0.0.html).

## [Unreleased]

Nothing here yet...

## [0.2.1] - 2022-03-22

### Added

- Explicit `@pyqtSlot` decorators
- Link to Qt5 docs in project docs with intersphinx
- Copyright comment in tests and `setup.py`
- Create version file for Windows compilation
- Create full compile.py script
- Add `Info.plist` file for macOS compilation
- Support --help and --version flags in `__main__.py`
- Create about dialog in help menu
- Implement minimum grid spacing

### Fixed

- Fix problems with compile script
- Fix small bugs and docstrings

## [0.2.0] - 2022-03-11

There were alpha tags before this, but I wasn't properly adhering to semantic versioning, so I'll start the
↪  changelog here.

If I'd been using semantic versioning from the start, there would much more changelog here, but instead, I'll just
↪  summarise the features.

### Added

- Matrix context with the `MatrixWrapper` class
- Parsing and evaluating matrix expressions
- A simple GUI with a viewport to render linear transformations
- Simple dialogs to create matrices and assign them to names
- Ability to render and animate linear transformations parsed from defined matrices
- Ability to zoom in and out of the viewport
- Add dialog to change display settings

[Unreleased]: https://github.com/DoctorDalek1963/lintrans/compare/v0.2.1...HEAD
[0.2.1]: https://github.com/DoctorDalek1963/lintrans/compare/v0.2.0...v0.2.1
[0.2.0]: https://github.com/DoctorDalek1963/lintrans/compare/13600cc6ff6299dc4a8101a367bc52fe08607554...v0.2.0
```

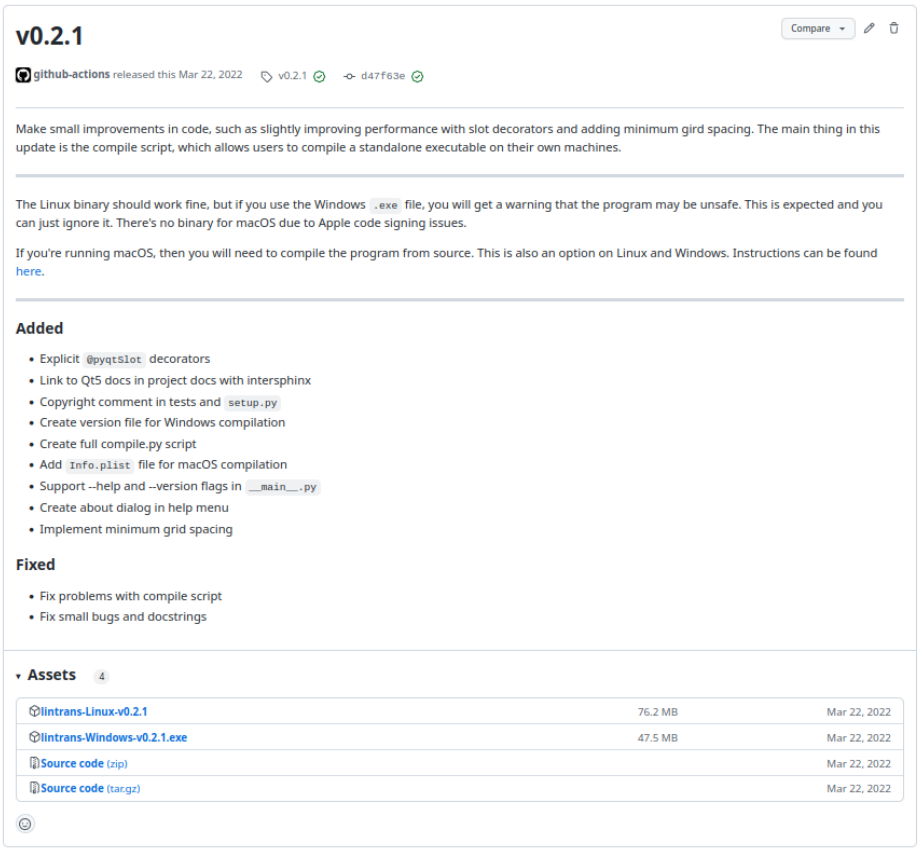


Figure 3.35: The release of `v0.2.1` on GitHub

3.9.12 Automating release note generation

I had to copy the changelog over and polish up the release notes myself for this release. It would be quite convenient to have a script that does this automatically, so I made one.

```
# 99a88575f9beb8fed2dcc41dacbb020b31bc8176
# generate_release_notes.py

"""A very simple script to generate release notes."""

import re
import sys

TEXT = '''DESCRIPTION

---

The Linux binary should work fine, but if you use the Windows `.exe` file, you will get a warning that the program may be unsafe. This is expected and you can just ignore it. There's no binary for macOS due to Apple code signing issues.

If you're running macOS, then you will need to compile the program from source. This is also an option on Linux and Windows. Instructions can be found [here](https://doctordalek1963.github.io/lintrans/tutorial/compile/).

---

CHANGELOG
'''

# This RegEx is complicated because of the newlines
# It requires the current tag to have a header like
```

```
    # ## [0.2.1] - 2022-03-22
    # And all other tags to have similar headers
    # It also won't work on the first tag, but that's fine
    RE_PATTERN = r'''(?<=## \[TAG_NAME\] - \d{4}-\d{2}-\d{2}

    ).*?(?=

    ## \[\d+\.\d+\.\d+(-[\S]+)?\] - \d{4}-\d{2}-\d{2})'''


    def main(args: list[str]) -> None:
        """Generate the release notes for this release and write them to `release_notes.md`."""
        if len(args) < 1:
            raise ValueError('Tag name is required to generate release notes')

        tag_name = args[0]

        print(f'Generating release notes for tag {tag_name}')

        with open('CHANGELOG.md', 'r', encoding='utf-8') as f:
            changelog_text = f.read()

        if (m := re.search(
            RE_PATTERN.replace('TAG_NAME', re.escape(tag_name[1:])),
            changelog_text,
            flags=re.S
        )) is not None:
            text = TEXT.replace('CHANGELOG', m.group(0))

        else:
            raise ValueError('Error in searching for changelog notes. Bad format')

        with open('release_notes.md', 'w', encoding='utf-8') as f:
            f.write(text)


    if __name__ == '__main__':
        main(sys.argv[1:])
```

This script just parses the changelog and generates a file called `release_notes.md`, which I can then automatically use as the body for the GitHub release by changing the workflow.

```
    # 99a88575f9beb8fed2dcc41dacbb020b31bc8176
    # .github/workflows/compile-release.yaml

    jobs:
...
      publish:
...
        steps:
...
          - name: Generate release notes
            run: python generate_release_notes.py $GITHUB_REF_NAME

          # This is practically the same step twice just to allow for pre-releases
          - name: Upload binaries (normal release)
            if: steps.checkprerelease.outputs.isprerelease == 0
            uses: softprops/action-gh-release@v1
            with:
              fail_on_unmatched_files: true
              prerelease: false
              draft: true
              body_path: release_notes.md
              files: dist/lintrans*

          - name: Upload binaries (pre-release)
            if: steps.checkprerelease.outputs.isprerelease == 1
            uses: softprops/action-gh-release@v1
            with:
              fail_on_unmatched_files: true
              prerelease: true
```

```
144             draft: true
145             body_path: release_notes.md
146             files: dist/lintrans*
```

3.10 Making `v0.2.2`

3.10.1 Hiding the background and transformed grids

I spoke to my main stakeholder, who is the teacher that will be using `lintrans` when it's finished, and she said that the background grid and transformed grid can get a little bit in the way of the core action and make it harder to understand what's happening. Taking this feedback on board, I decided to add a display setting to toggle the background grid, and one to toggle the transformed grid.

I did the background grid first and then repeated everything for the transformed version of the grid as well. I am combining them here for brevity. The first step was of course to add a display setting for each of them. Then I had to add checkboxes for them in the display settings dialog, and then incorporate the settings into the actual drawing of the canvas.

```
# d045057d568ac133b621ee9ca9daed361d570d7a
# src/lintrans/gui/settings.py

@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."""
```

```
# d045057d568ac133b621ee9ca9daed361d570d7a
# src/lintrans/gui/dialogs/settings.py

class DisplaySettingsDialog(SettingsDialog):
...
    def __init__(self, display_settings: DisplaySettings, *args, **kwargs):
...
        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
...
    def load_settings(self) -> None:
...
        self.checkbox_draw_background_grid.setChecked(self.display_settings.draw_background_grid)
        self.checkbox_draw_transformed_grid.setChecked(self.display_settings.draw_transformed_grid)
...
    def confirm_settings(self) -> None:
...
        self.display_settings.draw_background_grid = self.checkbox_draw_background_grid.isChecked()
        self.display_settings.draw_transformed_grid = self.checkbox_draw_transformed_grid.isChecked()
```

```
# d045057d568ac133b621ee9ca9daed361d570d7a
# src/lintrans/gui/plots/widgets.py
```

```
19  class VisualizeTransformationWidget(VectorGridPlot):
...
48      def paintEvent(self, event: QPaintEvent) -> None:
...
60          self.draw_background(painter, self.display_settings.draw_background_grid)
61
62          if self.display_settings.draw_transformed_grid:
63              self.draw_transformed_grid(painter)
```

```
# d045057d568ac133b621ee9ca9daed361d570d7a
# src/lintrans/gui/plots/classes.py
```

```
23  class BackgroundPlot(QWidget):
...
129     def draw_background(self, painter: QPainter, draw_grid: bool) -> None:
130         """Draw the background grid.
131
132         .. note:: This method is just a utility method for subclasses to use to render the background grid.
133
134         :param QPainter painter: The painter to draw the background with
135         :param bool draw_grid: Whether to draw the grid lines
136         """
137         if draw_grid:
138             painter.setPen(QPen(self.colour_background_grid, self.width_background_grid))
139
140             # Draw equally spaced vertical lines, starting in the middle and going out
141             # We loop up to half of the width. This is because we draw a line on each side in each iteration
142             for x in range(self.width() // 2 + self.grid_spacing, self.width(), self.grid_spacing):
143                 painter.drawLine(x, 0, x, self.height())
144                 painter.drawLine(self.width() - x, 0, self.width() - x, self.height())
145
146             # Same with the horizontal lines
147             for y in range(self.height() // 2 + self.grid_spacing, self.height(), self.grid_spacing):
148                 painter.drawLine(0, y, self.width(), y)
149                 painter.drawLine(0, self.height() - y, self.width(), self.height() - y)
150
151         # Now draw the axes
152         painter.setPen(QPen(self.colour_background_axes, self.width_background_grid))
153         painter.drawLine(self.width() // 2, 0, self.width() // 2, self.height())
154         painter.drawLine(0, self.height() // 2, self.width(), self.height() // 2)
```

Then I added this change to the changelog.

```
<!-- d045057d568ac133b621ee9ca9daed361d570d7a -->
<!-- CHANGELOG.md -->
```

```
12  ### Added
13
14  - Add options to hide background grid and transformed grid
```

3.10.2 Hiding the basis vectors

While I was implementing new display settings, I decided to implement hiding basis vectors. This will give users the option of just seeing the grid get transformed. The process was exactly the same as before. Add the setting, add it to the dialog, use it when drawing.

```
# 11ffbaf71f9fe29e1832a62f2b127aa3939e520d
# src/lintrans/gui/settings.py
```

```
15  class DisplaySettings:
...
29      draw_basis_vectors: bool = True
30      """This controls whether we want to draw the transformed basis vectors."""
```

```
# 11ffbaf71f9fe29e1832a62f2b127aa3939e520d
# src/lintrans/gui/dialogs/settings.py
```

```
70      class DisplaySettingsDialog(SettingsDialog):
...
73          def __init__(self, display_settings: DisplaySettings, *args, **kwargs):
...
103             self.checkbox_draw_basis_vectors = QCheckBox(self)
104             self.checkbox_draw_basis_vectors.setText('Draw basis &vectors')
105             self.checkbox_draw_basis_vectors.setToolTip(
106                 'Draw the transformed basis vectors'
107             )
108             self.dict_checkboxes['v'] = self.checkbox_draw_basis_vectors
...
212         def load_settings(self) -> None:
...
217             self.checkbox_draw_basis_vectors.setChecked(self.display_settings.draw_basis_vectors)
...
230         def confirm_settings(self) -> None:
...
235             self.display_settings.draw_basis_vectors = self.checkbox_draw_basis_vectors.isChecked()
```

```
# 11ffbaf71f9fe29e1832a62f2b127aa3939e520d
# src/lintrans/gui/plots/widgets.py

19      class VisualizeTransformationWidget(VectorGridPlot):
...
48          def paintEvent(self, event: QPaintEvent) -> None:
...
65              if self.display_settings.draw_basis_vectors:
66                  self.draw_basis_vectors(painter)
```

And then of course add it to the changelog.

```
<!-- 11ffbaf71f9fe29e1832a62f2b127aa3939e520d -->
<!-- CHANGELOG.md -->

12      ### Added
13
14      - Add options to hide background grid, transformed grid, and basis vectors
```

References

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

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

[3] Dyson 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] *About Code Signing*. Apple Inc. URL: https://developer.apple.com/library/archive/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html.

[8] *About `Info.plist` Keys and Values*. Apple Inc. URL: https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Introduction/Introduction.html#//apple_ref/doc/uid/TP40009248-SW1.

[9] *About Information Property List Files*. Apple Inc. URL: https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/AboutInformationPropertyListFiles.html#//apple_ref/doc/uid/TP40009254-SW1.

[10] *Apple Developer Program. What You Need To Enroll*. Apple Inc. URL: https://developer.apple.com/programs/enroll/.

[11] *Bundle Structures*. Apple Inc. URL: https://developer.apple.com/library/archive/documentation/CoreFoundation/Conceptual/CFBundles/BundleTypes/BundleTypes.html.

[12] *Fundamental theorem of algebra*. Wikipedia. URL: https://en.wikipedia.org/wiki/Fundamental_theorem_of_algebra.

[13] *GitHub Actions Documentation*. GitHub, Inc. URL: https://docs.github.com/en/actions.

[14] *GNU General Public License*. Version 3. Free Software Foundation. 29th June 2007. URL: https://www.gnu.org/licenses/gpl-3.0.en.html.

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

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

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

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

[19] Olivier Lacan. *keep a changelog*. Version 1.0.0. URL: https://keepachangelog.com/en/1.0.0/.

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

[21] *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.

[22] *NumPy `eig()` function*. Version 1.23. NumPy Developers. URL: https://numpy.org/doc/1.23/reference/generated/numpy.linalg.eig.html.

[23] *Nutika User Manual*. URL: https://nuitka.net/doc/user-manual.html.

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

[25] *PyInstaller Manual*. Version 4.10. URL: https://pyinstaller.org/en/v4.10/.

[26] *PyInstaller Manual. Capturing Windows Version Data*. Version 4.10. URL: https://pyinstaller.org/en/v4.10/usage.html#capturing-windows-version-data.

[27] *Pylint features (import graphs)*. Version 2.12.2. PyCQA. URL: https://pylint.readthedocs.io/en/v2.12.2/technical_reference/features.html?highlight=graph#imports-checker-options.

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

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

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

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

[32] *QValidator class*. The Qt Company. URL: https://doc.qt.io/qt-5/qvalidator.html.

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

[34] *QWidget Class (`baseSize` property)*. The Qt Company. URL: https://doc.qt.io/qt-5/qwidget.html#baseSize-prop.

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

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

[37] *QWidget Class (`size` property)*. The Qt Company. URL: https://doc.qt.io/qt-5/qwidget.html#size-prop.

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

[39] *Semantic Versioning*. Version 2.0.0. URL: https://semver.org/spec/v2.0.0.html.

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

[41] Brian Skinn. *`sphobjinv` documentation*. Version 2.2. URL: https://sphobjinv.readthedocs.io/en/v2.2/.

[42] *Sphinx. Python Documentation Generator*. The Sphinx developers. URL: https://www.sphinx-doc.org/en/master/.

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

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

[45] *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.

[46] *UPX. The Ultimate Packer for eXecutables*. URL: https://github.com/upx/upx.

[47] *Version Information Structures*. Microsoft. URL: https://learn.microsoft.com/en-gb/windows/win32/menurc/version-information-structures.

[48] *`VERSIONINFO` Resource*. Microsoft. URL: https://learn.microsoft.com/en-gb/windows/win32/menurc/versioninfo-resource.

A Project code

A.1 `global_settings.py`

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

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

"""This module provides the :class:`GlobalSettings` class, which is used to access global settings."""

from __future__ import annotations

import os
import pathlib
import pickle
import subprocess
import sys
from copy import copy
from dataclasses import dataclass
from enum import Enum
from pathlib import Path
from typing import Optional, Tuple

from singleton_decorator import singleton

import lintrans

UpdateType = Enum('UpdateType', 'auto prompt never')
"""An enum of possible update prompt types."""


@dataclass(slots=True)
class GlobalSettingsData:
    """A simple dataclass to store the configurable data of the global settings."""

    update_type: UpdateType = UpdateType.prompt
    """This is the desired type of update prompting."""

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

    snap_dist: float = 0.1
    """This is the distance in grid coords that the cursor needs to be from an integer point to snap to it."""

    snap_to_int_coords: bool = True
    """This decides whether or not vectors should snap to integer coordinates when being dragged around."""

    def save_to_file(self, filename: str) -> None:
        """Save the global settings data to a file, creating parent directories as needed."""
        parent_dir = pathlib.Path(os.path.expanduser(filename)).parent.absolute()

        if not os.path.isdir(parent_dir):
            os.makedirs(parent_dir)

        data: Tuple[str, GlobalSettingsData] = (lintrans.__version__, self)

        with open(filename, 'wb') as f:
            pickle.dump(data, f, protocol=4)

    @classmethod
    def load_from_file(cls, filename: str) -> Tuple[str, GlobalSettingsData]:
        """Return the global settings data that was previously saved to ``filename`` along with some extra
    ↪  information.

        The tuple we return has the version of lintrans that was used to save the file, and the data itself.

        :raises EOFError: If the file doesn't contain a pickled Python object
        :raises FileNotFoundError: If the file doesn't exist
        :raises ValueError: If the file contains a pickled object of the wrong type
```

```
        """
        if not os.path.isfile(filename):
            return lintrans.__version__, cls()

        with open(filename, 'rb') as f:
            file_data = pickle.load(f)

        if not isinstance(file_data, tuple):
            raise ValueError(f'File {filename} contains pickled object of the wrong type (must be tuple)')

        # Create a default object and overwrite the fields that we have
        data = cls()
        for attr in file_data[1].__slots__:
            # Try to get the attribute from the old data, but don't worry if we can't,
            # because that means it's from an older version, so we can use the default
            # values from `cls()`
            try:
                setattr(data, attr, getattr(file_data[1], attr))
            except AttributeError:
                pass

        return file_data[0], data


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

    .. note::
       This is a singleton class because we only want :meth:`__init__` to be called once
       to reduce processing time. We also can't cache it as a global variable because that
       would be created at import time, leading to infinite process recursion when lintrans
       tries to call its own executable to find out if it's compiled or interpreted.

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

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

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

        else:
            # This should be unreachable because the only other option for os.name is 'java'
            # for Jython, but Jython only supports Python 2.7, which has been EOL for a while
            # lintrans is only compatible with Python >= 3.10 anyway
            raise OSError(f'Unrecognised OS "{os.name}"')

        sub_directories = ['saves', 'crash_reports']

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

        self._executable_path: Optional[str] = None

        self._settings_file = os.path.join(self._directory, 'settings.dat')
        self._display_settings_file = os.path.join(self._directory, 'display_settings.dat')

        try:
            self._data = GlobalSettingsData.load_from_file(self._settings_file)[1]
```

```
            except KeyError:
                self._data = GlobalSettingsData()
                self._data.save_to_file(self._settings_file)

        def get_executable_path(self) -> str:
            """Return the path to the binary executable, or an empty string if lintrans is not installed standalone.

            This method will call :attr:`sys.executable` to see if it's lintrans. If it is, then we cache the path for
            future use and return it. Otherwise, it's a Python interpreter, so we return an empty string instead.
            """
            if self._executable_path is None:
                executable_path = sys.executable
                if os.path.isfile(executable_path):
                    version_output = subprocess.run(
                        [executable_path, '--version'],
                        stdout=subprocess.PIPE,
                        shell=(os.name == 'nt')
                    ).stdout.decode()

                    if 'lintrans' in version_output:
                        self._executable_path = executable_path
                    else:
                        self._executable_path = ''

            return self._executable_path or ''

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

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

        def get_settings_file(self) -> str:
            """Return the full path of the settings file."""
            return self._settings_file

        def save_display_settings(self, settings: lintrans.gui.settings.DisplaySettings) -> None:
            """Save the given display settings to the default file."""
            settings.save_to_file(self._display_settings_file)

        def get_display_settings(self) -> lintrans.gui.settings.DisplaySettings:
            """Get the display settings from the default file, using the defaults for anything that's not available."""
            return lintrans.gui.settings.DisplaySettings.load_from_file(self._display_settings_file)[1]

        def get_update_download_filename(self) -> str:
            """Return a name for a temporary file next to the executable.

            This method is used when downloading a new version of lintrans into a temporary file.
            This is needed to allow :func:`os.rename` instead of :func:`shutil.move`. The first
            requires the src and dest to be on the same partition, but also allows us to replace
            the running executable.
            """
            return str(Path(self.get_executable_path()).parent / 'lintrans-update-temp.dat')

        def get_update_replace_bat_filename(self) -> str:
            """Return the full path of the ``replace.bat`` file needed to update on Windows.

            See :meth:`get_update_download_filename`.
            """
            return str(Path(self.get_executable_path()).parent / 'replace.bat')

        def get_data(self) -> GlobalSettingsData:
            """Return a copy of the internal global settings data."""
            return copy(self._data)

        def set_data(self, data: GlobalSettingsData) -> None:
            """Set the internal global settings data and save it to a file."""
            self._data = data
            self._data.save_to_file(self._settings_file)

        def set_update_type(self, type_: UpdateType) -> None:
```

```
213             """Set the internal data update type."""
214             data = self.get_data()
215             data.update_type = type_
216             self.set_data(data)
```

A.2 __main__.py

```
1   #!/usr/bin/env python
2
3   # lintrans - The linear transformation visualizer
4   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
5
6   # This program is licensed under GNU GPLv3, available here:
7   # <https://www.gnu.org/licenses/gpl-3.0.html>
8
9   """This module provides a :func:`main` function to interpret command line arguments and run the program."""
10
11  from argparse import ArgumentParser
12  from textwrap import dedent
13
14  from lintrans import __version__, gui
15  from lintrans.crash_reporting import set_excepthook, set_signal_handler
16
17
18  def main() -> None:
19      """Interpret program-specific command line arguments and run the main window in most cases.
20
21      If the user supplies ``--help`` or ``--version``, then we simply respond to that and then return.
22      If they don't supply either of these, then we run :func:`lintrans.gui.main_window.main`.
23
24      :param List[str] args: The full argument list (including program name)
25      """
26      parser = ArgumentParser(add_help=False)
27
28      parser.add_argument(
29          'filename',
30          nargs='?',
31          type=str,
32          default=None
33      )
34
35      parser.add_argument(
36          '-h',
37          '--help',
38          default=False,
39          action='store_true'
40      )
41
42      parser.add_argument(
43          '-V',
44          '--version',
45          default=False,
46          action='store_true'
47      )
48
49      parsed_args = parser.parse_args()
50
51      if parsed_args.help:
52          print(dedent('''
53          Usage: lintrans [option] [filename]
54
55          Arguments:
56              filename         The name of a session file to open
57
58          Options:
59              -h, --help       Display this help text and exit
60              -V, --version    Display the version information and exit''')[1:])
61          return
62
63      if parsed_args.version:
```

```
64              print(dedent(f'''
65              lintrans (version {__version__})
66              The linear transformation visualizer
67
68              Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
69
70              This program is licensed under GNU GPLv3, available here:
71              <https://www.gnu.org/licenses/gpl-3.0.html>'''[1:]))
72              return
73
74          gui.main(parsed_args.filename)
75
76
77      if __name__ == '__main__':
78          set_excepthook()
79          set_signal_handler()
80          main()
```

A.3 `__init__.py`

```
1       # lintrans - The linear transformation visualizer
2       # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4       # This program is licensed under GNU GPLv3, available here:
5       # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7       """This is the top-level ``lintrans`` package, which contains all the subpackages of the project."""
8
9       from . import (crash_reporting, global_settings, gui, matrices, typing_,
10                     updating)
11
12      __version__ = '0.4.2-alpha'
13
14      __all__ = ['crash_reporting', 'global_settings', 'gui', 'matrices', 'typing_', 'updating', '__version__']
```

A.4 `crash_reporting.py`

```
1       # lintrans - The linear transformation visualizer
2       # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4       # This program is licensed under GNU GPLv3, available here:
5       # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7       """This module provides functions to report crashes and log them.
8
9       The only functions you should be calling directly are :func:`set_excepthook`
10      and :func:`set_signal_handler` to setup handlers for unhandled exceptions
11      and unhandled operating system signals respectively.
12      """
13
14      from __future__ import annotations
15
16      import os
17      import platform
18      import signal
19      import sys
20      from datetime import datetime
21      from signal import SIGABRT, SIGFPE, SIGILL, SIGSEGV, SIGTERM
22      from textwrap import indent
23      from types import FrameType, TracebackType
24      from typing import NoReturn, Type
25
26      from PyQt5.QtCore import PYQT_VERSION_STR, QT_VERSION_STR
27      from PyQt5.QtWidgets import QApplication
28
29      import lintrans
30      from lintrans.typing_ import is_matrix_type
31
```

```
32      from .global_settings import GlobalSettings
33      from .gui.main_window import LintransMainWindow
34
35
36      def _get_datetime_string() -> str:
37          """Get the date and time as a string with a space in the middle."""
38          return datetime.now().strftime('%Y-%m-%d %H:%M:%S')
39
40
41      def _get_main_window() -> LintransMainWindow:
42          """Return the only instance of :class:`~lintrans.gui.main_window.LintransMainWindow`.
43
44          :raises RuntimeError: If there is not exactly 1 instance of
    ↪     :class:`~lintrans.gui.main_window.LintransMainWindow`
45          """
46          widgets = [
47              x for x in QApplication.topLevelWidgets()
48              if isinstance(x, LintransMainWindow)
49          ]
50
51          if len(widgets) != 1:
52              raise RuntimeError(f'Expected 1 widget of type LintransMainWindow but found {len(widgets)}')
53
54          return widgets[0]
55
56
57      def _get_system_info() -> str:
58          """Return a string of all the system we could gather."""
59          info = 'SYSTEM INFO:\n'
60
61          info += f'  lintrans: {lintrans.__version__}\n'
62          info += f'  Python: {platform.python_version()}\n'
63          info += f'  Qt5: {QT_VERSION_STR}\n'
64          info += f'  PyQt5: {PYQT_VERSION_STR}\n'
65          info += f'  Platform: {platform.platform()}\n'
66
67          info += '\n'
68          return info
69
70
71      def _get_error_origin(
72          *,
73          exc_type: Type[BaseException] | None,
74          exc_value: BaseException | None,
75          traceback: TracebackType | None,
76          signal_number: int | None,
77          stack_frame: FrameType | None
78      ) -> str:
79          """Return a string specifying the full origin of the error, as best as we can determine.
80
81          This function has effectively two signatures. If the fatal error is caused by an exception,
82          then the first 3 arguments will be used to match the signature of :func:`sys.excepthook`.
83          If it's caused by a signal, then the last two will be used to match the signature of the
84          handler in :func:`signal.signal`. This function should never be used outside this file, so
85          we don't account for a mixture of arguments.
86
87          :param exc_type: The type of the exception that caused the crash
88          :param exc_value: The value of the exception itself
89          :param traceback: The traceback object
90          :param signal_number: The number of the signal that caused the crash
91          :param stack_frame: The current stack frame object
92
93          :type exc_type: Type[BaseException] | None
94          :type exc_value: BaseException | None
95          :type traceback: types.TracebackType | None
96          :type signal_number: int | None
97          :type stack_frame: types.FrameType | None
98          """
99          origin = 'CRASH ORIGIN:\n'
100
101         if exc_type is not None and exc_value is not None and traceback is not None:
102             # We want the frame where the exception actually occurred, so we have to descend the traceback
103             # I don't know why we aren't given this traceback in the first place
```

```
104             tb = traceback
105             while tb.tb_next is not None:
106                 tb = tb.tb_next
107
108             frame = tb.tb_frame
109
110             origin += f'  Exception "{exc_value}"\n  of type {exc_type.__name__} in call to {frame.f_code.co_name}()\n'
                ↪  \
111                 f'  on line {frame.f_lineno} of {frame.f_code.co_filename}'
112
113         elif signal_number is not None and stack_frame is not None:
114             origin += f'  Signal "{signal.strsignal(signal_number)}" received in call to
                ↪  {stack_frame.f_code.co_name}()\n' \
115                 f'  on line {stack_frame.f_lineno} of {stack_frame.f_code.co_filename}'
116
117         else:
118             origin += '  UNKNOWN (not exception or signal)'
119
120         origin += '\n\n'
121
122         return origin
123
124
125     def _get_display_settings() -> str:
126         """Return a string representing all of the display settings."""
127         raw_settings = _get_main_window()._plot.display_settings
128         display_settings = {
129             k: getattr(raw_settings, k)
130             for k in raw_settings.__slots__
131             if not k.startswith('_')
132         }
133
134         string = 'Display settings:\n'
135
136         for setting, value in display_settings.items():
137             string += f'  {setting}: {value}\n'
138
139         return string
140
141
142     def _get_post_mortem() -> str:
143         """Return whatever post mortem data we could gather from the window."""
144         window = _get_main_window()
145
146         try:
147             matrix_wrapper = window._matrix_wrapper
148             expression_history = window._expression_history
149             exp_hist_index = window._expression_history_index
150             plot = window._plot
151             point_i = plot.point_i
152             point_j = plot.point_j
153
154         except (AttributeError, RuntimeError) as e:
155             return f'UNABLE TO GET POST MORTEM DATA:\n  {e!r}\n'
156
157         post_mortem = 'Matrix wrapper:\n'
158
159         for matrix_name, matrix_value in matrix_wrapper.get_defined_matrices():
160             post_mortem += f'  {matrix_name}: '
161
162             if is_matrix_type(matrix_value):
163                 post_mortem += f'[{matrix_value[0][0]} {matrix_value[0][1]}; {matrix_value[1][0]} {matrix_value[1][1]}]'
164             else:
165                 post_mortem += f'"{matrix_value}"'
166
167             post_mortem += '\n'
168
169         post_mortem += f'\nExpression box: "{window._lineedit_expression_box.text()}"'
170         post_mortem += f'\nCurrently displayed: [{point_i[0]} {point_j[0]}; {point_i[1]} {point_j[1]}]'
171         post_mortem += f'\nAnimating (sequence): {window._animating} ({window._animating_sequence})\n'
172
173         post_mortem += f'\nExpression history (index={exp_hist_index}):'
174         post_mortem += '\n  ['
```

```
        for item in expression_history:
            post_mortem += f'\n    {item!r},'
        post_mortem += '\n  ]\n'

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

        post_mortem += '\n' + _get_display_settings()

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


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

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

        return report


    def _report_crash(
        *,
        exc_type: Type[BaseException] | None = None,
        exc_value: BaseException | None = None,
        traceback: TracebackType | None = None,
        signal_number: int | None = None,
        stack_frame: FrameType | None = None
    ) -> NoReturn:
        """Generate a crash report and write it to a log file and stderr.

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

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

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

        sys.exit(255)


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

```
            traceback: TracebackType | None
        ) -> None:
            _report_crash(exc_type=exc_type, exc_value=exc_value, traceback=traceback)

        sys.excepthook = _custom_excepthook


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

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

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

A.5 `updating.py`

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

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

    """This module provides functions for updating the lintrans executable in a proper installation.

    If the user is using a standalone executable for lintrans, then we don't know where it is and
    we therefore can't update it.
    """

    from __future__ import annotations

    import os
    import re
    import subprocess
    from threading import Thread
    from typing import Optional, Tuple
    from urllib.error import URLError
    from urllib.request import urlopen

    from packaging import version

    from lintrans.global_settings import GlobalSettings


    def new_version_exists() -> Tuple[bool, Optional[str]]:
        """Check if the latest version of lintrans is newer than the current version.

        This function either returns (False, None) or (True, str) where the string is the new version.

        .. note::
           This function will default to False if it can't get the current or latest version, or if
           :meth:`~lintrans.global_settings.GlobalSettings.get_executable_path` returns ''
           (probablybecause lintrans is being run as a Python package)

           However, it will return True if the executable path is defined but the executable doesn't actually exist.

           This last behaviour is mostly to make testing easier by spoofing
           :meth:`~lintrans.global_settings.GlobalSettings.get_executable_path`.
        """
        executable_path = GlobalSettings().get_executable_path()
        if executable_path == '':
            return False, None
```

```
        try:
            html: str = urlopen('https://github.com/DoctorDalek1963/lintrans/releases/latest').read().decode()
        except (UnicodeDecodeError, URLError):
            return False, None

        match = re.search(
            r'(?<=DoctorDalek1963/lintrans/releases/tag/v)\d+\.\d+\.\d+(?=;)',
            html
        )
        if match is None:
            return False, None

        latest_version_str = match.group(0)
        latest_version = version.parse(latest_version_str)

        # If the executable doesn't exist, then we definitely want to update it
        if not os.path.isfile(executable_path):
            return True, latest_version_str

        # Now check the current version
        version_output = subprocess.run(
            [executable_path, '--version'],
            stdout=subprocess.PIPE,
            shell=(os.name == 'nt')
        ).stdout.decode()

        match = re.search(r'(?<=lintrans \(version )\d+\.\d+\.\d+(-\w+(-?\d+))?(?=\))', version_output)

        if match is None:
            return False, None

        current_version = version.parse(match.group(0))

        if latest_version > current_version:
            return True, latest_version_str

        return False, None


    def update_lintrans() -> None:
        """Update the lintrans binary executable, failing silently.

        This function only makes sense if lintrans was installed, rather than being used as an executable.
        We ask the :class:`~lintrans.global_settings.GlobalSettings` singleton where the executable is and,
        if it exists, then we replace the old executable with the new one. This means that the next time
        lintrans gets run, it will use the most recent version.

        .. note::
           This function doesn't care if the latest version on GitHub is actually newer than the current
           version. Use :func:`new_version_exists` to check.
        """
        executable_path = GlobalSettings().get_executable_path()
        if executable_path == '':
            return

        try:
            html: str = urlopen('https://github.com/DoctorDalek1963/lintrans/releases/latest').read().decode()
        except (UnicodeDecodeError, URLError):
            return

        match = re.search(
            r'(?<=DoctorDalek1963/lintrans/releases/tag/v)\d+\.\d+\.\d+(?=;)',
            html
        )
        if match is None:
            return

        latest_version = version.parse(match.group(0))

        # We now know that the latest version is newer, and where the executable is,
        # so we can begin the replacement process
        url = 'https://github.com/DoctorDalek1963/lintrans/releases/download/'

```

```
        if os.name == 'posix':
            url += f'v{latest_version}/lintrans-Linux-{latest_version}'

        elif os.name == 'nt':
            url += f'v{latest_version}/lintrans-Windows-{latest_version}.exe'

        else:
            return

        temp_file = GlobalSettings().get_update_download_filename()

        # If the temp file already exists, then another instance of lintrans (probably
        # in a background thread) is currently updating, so we don't want to interfere
        if os.path.isfile(temp_file):
            return

        with open(temp_file, 'wb') as f:
            try:
                f.write(urlopen(url).read())
            except URLError:
                return

        if os.name == 'posix':
            os.rename(temp_file, executable_path)
            subprocess.run(['chmod', '+x', executable_path])

        elif os.name == 'nt':
            # On Windows, we need to leave a process running in the background to automatically
            # replace the exe file when lintrans stops running
            script = '@echo off\n' \
                ':loop\n\n' \
                'timeout 5 >nul\n' \
                'tasklist /fi "IMAGENAME eq lintrans.exe" /fo csv 2>nul | find /I "lintrans.exe" >nul\n' \
                'if "%ERRORLEVEL%"=="0" goto :loop\n\n' \
                f'del "{executable_path}"\n' \
                f'rename "{temp_file}" lintrans.exe\n\n' \
                'start /b "" cmd /c del "%~f0"&exit /b'

            replace_bat = GlobalSettings().get_update_replace_bat_filename()
            with open(replace_bat, 'w', encoding='utf-8') as f:
                f.write(script)

            subprocess.Popen(['start', '/min', replace_bat], shell=True)


    def update_lintrans_in_background(*, check: bool) -> None:
        """Use multithreading to run :func:`update_lintrans` in the background."""
        def func() -> None:
            if check:
                if new_version_exists()[0]:
                    update_lintrans()
            else:
                update_lintrans()

        p = Thread(target=func)
        p.start()
```

A.6 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 nptyping import Float, NDArray, Shape
from numpy import ndarray

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

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

MatrixType: TypeAlias = NDArray[Shape['2, 2'], Float]
"""This type represents a 2x2 matrix as a NumPy array."""

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

MatrixParseList: TypeAlias = 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[MatrixType]:
    """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:`MatrixType`.
    """
    return isinstance(matrix, ndarray) and matrix.shape == (2, 2)
```

A.7 matrices/wrapper.py

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

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

"""This module contains the main :class:`MatrixWrapper` class and a function to create a matrix from an angle."""

from __future__ import annotations

import re
from copy import copy
from functools import reduce
from operator import add, matmul
from typing import Any, Dict, List, Optional, Set, Tuple, Union

import numpy as np

from lintrans.typing_ import MatrixType, is_matrix_type

from .parse import (get_matrix_identifiers, parse_matrix_expression,
                    validate_matrix_expression)
from .utility import create_rotation_matrix

```

```
25      _ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
26
27
28      class MatrixWrapper:
29          """A wrapper class to hold all possible matrices and allow access to them.
30
31          .. note::
32             When defining a custom matrix, its name must be a capital letter and cannot be ``I``.
33
34          The contained matrices can be accessed and assigned to using square bracket notation.
35
36          :Example:
37
38          >>> wrapper = MatrixWrapper()
39          >>> wrapper['I']
40          array([[1., 0.],
41                 [0., 1.]])
42          >>> wrapper['M']  # Returns None
43          >>> wrapper['M'] = np.array([[1, 2], [3, 4]])
44          >>> wrapper['M']
45          array([[1., 2.],
46                 [3., 4.]])
47          """
48
49          def __init__(self):
50              """Initialize a :class:`MatrixWrapper` object with a dictionary of matrices which can be accessed."""
51              self._matrices: Dict[str, Optional[Union[MatrixType, str]]] = {
52                  'A': None, 'B': None, 'C': None, 'D': None,
53                  'E': None, 'F': None, 'G': None, 'H': None,
54                  'I': np.eye(2),  # I is always defined as the identity matrix
55                  'J': None, 'K': None, 'L': None, 'M': None,
56                  'N': None, 'O': None, 'P': None, 'Q': None,
57                  'R': None, 'S': None, 'T': None, 'U': None,
58                  'V': None, 'W': None, 'X': None, 'Y': None,
59                  'Z': None
60              }
61
62          def __repr__(self) -> str:
63              """Return a nice string repr of the :class:`MatrixWrapper` for debugging."""
64              defined_matrices = ''.join([k for k, v in self._matrices.items() if v is not None])
65              return f'<{self.__class__.__module__}.{self.__class__.__name__} object with ' \
66                     f"{len(defined_matrices)} defined matrices: '{defined_matrices}'>"
67
68          def __eq__(self, other: Any) -> bool:
69              """Check for equality in wrappers by comparing dictionaries.
70
71              :param Any other: The object to compare this wrapper to
72              """
73              if not isinstance(other, self.__class__):
74                  return NotImplemented
75
76              # We loop over every matrix and check if every value is equal in each
77              for name in self._matrices:
78                  s_matrix = self[name]
79                  o_matrix = other[name]
80
81                  if s_matrix is None and o_matrix is None:
82                      continue
83
84                  elif (s_matrix is None and o_matrix is not None) or \
85                       (s_matrix is not None and o_matrix is None):
86                      return False
87
88                  # This is mainly to satisfy mypy, because we know these must be matrices
89                  elif not is_matrix_type(s_matrix) or not is_matrix_type(o_matrix):
90                      return False
91
92                  # Now we know they're both NumPy arrays
93                  elif np.array_equal(s_matrix, o_matrix):
94                      continue
95
96                  else:
97                      return False
```

```
98
99              return True
100
101         def __hash__(self) -> int:
102             """Return the hash of the matrices dictionary."""
103             return hash(self._matrices)
104
105         def __getitem__(self, name: str) -> Optional[MatrixType]:
106             """Get the matrix with the given identifier.
107
108             If it is a simple name, it will just be fetched from the dictionary. If the identifier is ``rot(x)``, with
109             a given angle in degrees, then we return a new matrix representing a rotation by that angle. If the
    ↪  identifier
110             is something like ``[1 2;3 4]``, then we will evaluate this matrix (we assume it will have whitespace
    ↪  exactly
111             like the example; see :func:`lintrans.matrices.parse.strip_whitespace`).
112
113             .. note::
114                If the named matrix is defined as an expression, then this method will return its evaluation.
115                If you want the expression itself, use :meth:`get_expression`.
116
117             :param str name: The name of the matrix to get
118             :returns Optional[MatrixType]: The value of the matrix (could be None)
119
120             :raises NameError: If there is no matrix with the given name
121             """
122             # Return a new rotation matrix
123             if (match := re.match(r'^rot\((-?\d*\.?\d*)\)$', name)) is not None:
124                 return create_rotation_matrix(float(match.group(1)))
125
126             if (match := re.match(
127                     r'\[(-?\d+(?:\.\d+)?) (-?\d+(?:\.\d+)?);(-?\d+(?:\.\d+)?) (-?\d+(?:\.\d+)?)\]',
128                     name
129             )) is not None:
130                 a = float(match.group(1))
131                 b = float(match.group(2))
132                 c = float(match.group(3))
133                 d = float(match.group(4))
134                 return np.array([[a, b], [c, d]])
135
136             if name not in self._matrices:
137                 if validate_matrix_expression(name):
138                     return self.evaluate_expression(name)
139
140                 raise NameError(f'Unrecognised matrix name "{name}"')
141
142             # We copy the matrix before we return it so the user can't accidentally mutate the matrix
143             matrix = copy(self._matrices[name])
144
145             if isinstance(matrix, str):
146                 return self.evaluate_expression(matrix)
147
148             return matrix
149
150         def __setitem__(self, name: str, new_matrix: Optional[Union[MatrixType, str]]) -> None:
151             """Set the value of matrix ``name`` with the new_matrix.
152
153             The new matrix may be a simple 2x2 NumPy array, or it could be a string, representing an
154             expression in terms of other, previously defined matrices.
155
156             :param str name: The name of the matrix to set the value of
157             :param Optional[Union[MatrixType, str]] new_matrix: The value of the new matrix (could be None)
158
159             :raises NameError: If the name isn't a legal matrix name
160             :raises TypeError: If the matrix isn't a valid 2x2 NumPy array or expression in terms of other defined
    ↪  matrices
161             :raises ValueError: If you attempt to define a matrix in terms of itself
162             """
163             if not (name in self._matrices and name != 'I'):
164                 raise NameError('Matrix name is illegal')
165
166             if new_matrix is None:
167                 self._matrices[name] = None
```

```
            return

        if isinstance(new_matrix, str):
            if self.is_valid_expression(new_matrix):
                if name not in new_matrix and \
                        name not in self.get_expression_dependencies(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_matrix_dependencies(self, matrix_name: str) -> Set[str]:
        """Return all the matrices (as identifiers) that the given matrix (indirectly) depends on.

        If A depends on nothing, B directly depends on A, and C directly depends on B,
        then we say C depends on B `and` A.
        """
        expression = self.get_expression(matrix_name)
        if expression is None:
            return set()

        s = set()
        identifiers = get_matrix_identifiers(expression)
        for identifier in identifiers:
            s.add(identifier)
            s.update(self.get_matrix_dependencies(identifier))

        return s

    def get_expression_dependencies(self, expression: str) -> Set[str]:
        """Return all the matrices that the given expression depends on.

        This method just calls :meth:`get_matrix_dependencies` on each matrix
        identifier in the expression. See that method for details.

        If an expression contains a matrix that has no dependencies, then the
        expression is `not` considered to depend on that matrix. But it `is`
        considered to depend on any matrix that has its own dependencies.
        """
        s = set()
        for iden in get_matrix_identifiers(expression):
            s.update(self.get_matrix_dependencies(iden))
        return s

    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:
```

```
241             """Check if the given expression is valid, using the context of the wrapper.
242
243             This method calls :func:`lintrans.matrices.parse.validate_matrix_expression`, but also
244             ensures that all the matrices in the expression are defined in the wrapper.
245
246             :param str expression: The expression to validate
247             :returns bool: Whether the expression is valid in this wrapper
248
249             :raises LinAlgError: If a matrix is defined in terms of the inverse of a singular matrix
250             """
251             # Get rid of the transposes to check all capital letters
252             new_expression = expression.replace('^T', '').replace('^{T}', '')
253
254             # Make sure all the referenced matrices are defined
255             for matrix in [x for x in new_expression if re.match('[A-Z]', x)]:
256                 if self[matrix] is None:
257                     return False
258
259                 if (expr := self.get_expression(matrix)) is not None:
260                     if not self.is_valid_expression(expr):
261                         return False
262
263             return validate_matrix_expression(expression)
264
265         def evaluate_expression(self, expression: str) -> MatrixType:
266             """Evaluate a given expression and return the matrix evaluation.
267
268             :param str expression: The expression to be parsed
269             :returns MatrixType: The matrix result of the expression
270
271             :raises ValueError: If the expression is invalid
272             """
273             if not self.is_valid_expression(expression):
274                 raise ValueError('The expression is invalid')
275
276             parsed_result = parse_matrix_expression(expression)
277             final_groups: List[List[MatrixType]] = []
278
279             for group in parsed_result:
280                 f_group: List[MatrixType] = []
281
282                 for multiplier, identifier, index in group:
283                     if index == 'T':
284                         m = self[identifier]
285
286                         # This assertion is just so mypy doesn't complain
287                         # We know this won't be None, because we know that this matrix is defined in this wrapper
288                         assert m is not None
289                         matrix_value = m.T
290
291                     else:
292                         # Again, this assertion is just for mypy
293                         # We know this will be a matrix, but since upgrading from NumPy 1.21 to 1.23
294                         # (to fix a bug with GH Actions on Windows), mypy complains about matrix_power()
295                         base_matrix = self[identifier]
296                         assert is_matrix_type(base_matrix)
297
298                         matrix_value = np.linalg.matrix_power(base_matrix, 1 if index == '' else int(index))
299
300                     matrix_value *= 1 if multiplier == '' else float(multiplier)
301                     f_group.append(matrix_value)
302
303                 final_groups.append(f_group)
304
305             return reduce(add, [reduce(matmul, group) for group in final_groups])
306
307         def get_defined_matrices(self) -> List[Tuple[str, Union[MatrixType, str]]]:
308             """Return a list of tuples containing the name and value of all defined matrices in the wrapper.
309
310             :returns: A list of tuples where the first element is the name, and the second element is the value
311             :rtype: List[Tuple[str, Union[MatrixType, str]]]
312             """
313             matrices = []
```

```
314
315             for name, value in self._matrices.items():
316                 if value is not None:
317                     matrices.append((name, value))
318
319             return matrices
320
321         def undefine_matrix(self, name: str) -> Set[str]:
322             """Safely undefine the given matrix by also undefining any matrices that depend on it."""
323             if not (name in self._matrices and name != 'I'):
324                 raise NameError('Matrix name is illegal')
325
326             # This maps each matrix to all the matrices that depend on it
327             dependents_map = {
328                 x: set(y for y in _ALPHABET_NO_I if x in self.get_matrix_dependencies(y))
329                 for x in _ALPHABET_NO_I
330             }
331
332             s: Set[str] = set(name)
333             self[name] = None
334             for x in dependents_map[name]:
335                 s.update(self.undefine_matrix(x))
336
337             return s
```

A.8 `matrices/utility.py`

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides simple utility methods for matrix and vector manipulation."""
8
9   from __future__ import annotations
10
11  import math
12  from typing import Tuple
13
14  import numpy as np
15
16  from lintrans.typing_ import MatrixType
17
18
19  def polar_coords(x: float, y: float, *, degrees: bool = False) -> Tuple[float, float]:
20      r"""Return the polar coordinates of a given (x, y) Cartesian coordinate.
21
22      .. note:: We're returning the angle in the range :math:`[0, 2\pi)`
23      """
24      radius = math.hypot(x, y)
25
26      # PyCharm complains about np.angle taking a complex argument even though that's what it's designed for
27      # noinspection PyTypeChecker
28      angle = float(np.angle(x + y * 1j, degrees))
29
30      if angle < 0:
31          angle += 2 * np.pi
32
33      return radius, angle
34
35
36  def rect_coords(radius: float, angle: float, *, degrees: bool = False) -> Tuple[float, float]:
37      """Return the rectilinear coordinates of a given polar coordinate."""
38      if degrees:
39          angle = np.radians(angle)
40
41      return radius * np.cos(angle), radius * np.sin(angle)
42
43
```

```python
def rotate_coord(x: float, y: float, angle: float, *, degrees: bool = False) -> Tuple[float, float]:
    """Rotate a rectilinear coordinate by the given angle."""
    if degrees:
        angle = np.radians(angle)

    r, theta = polar_coords(x, y, degrees=degrees)
    theta = (theta + angle) % (2 * np.pi)

    return rect_coords(r, theta, degrees=degrees)


def create_rotation_matrix(angle: float, *, degrees: bool = True) -> MatrixType:
    """Create a matrix representing a rotation (anticlockwise) by the given angle.

    :Example:

    >>> create_rotation_matrix(30)
    array([[ 0.8660254, -0.5      ],
           [ 0.5      ,  0.8660254]])
    >>> create_rotation_matrix(45)
    array([[ 0.70710678, -0.70710678],
           [ 0.70710678,  0.70710678]])
    >>> create_rotation_matrix(np.pi / 3, degrees=False)
    array([[ 0.5      , -0.8660254],
           [ 0.8660254,  0.5      ]])

    :param float angle: The angle to rotate anticlockwise by
    :param bool degrees: Whether to interpret the angle as degrees (True) or radians (False)
    :returns MatrixType: The resultant matrix
    """
    rad = np.deg2rad(angle % 360) if degrees else angle % (2 * np.pi)
    return np.array([
        [np.cos(rad), -1 * np.sin(rad)],
        [np.sin(rad), np.cos(rad)]
    ])


def is_valid_float(string: str) -> bool:
    """Check if the string is a valid float (or anything that can be cast to a float, such as an int).

    This function simply checks that ``float(string)`` doesn't raise an error.

    .. note:: An empty string is not a valid float, so will return False.

    :param str string: The string to check
    :returns bool: Whether the string is a valid float
    """
    try:
        float(string)
        return True
    except ValueError:
        return False


def round_float(num: float, precision: int = 5) -> str:
    """Round a floating point number to a given number of decimal places for pretty printing.

    :param float num: The number to round
    :param int precision: The number of decimal places to round to
    :returns str: The rounded number for pretty printing
    """
    # Round to ``precision`` number of decimal places
    string = str(round(num, precision))

    # Cut off the potential final zero
    if string.endswith('.0'):
        return string[:-2]

    elif 'e' in string:  # Scientific notation
        split = string.split('e')
        # The leading 0 only happens when the exponent is negative, so we know there'll be a minus sign
        return split[0] + 'e-' + split[1][1:].lstrip('0')

```

```
117         else:
118             return string
```

A.9 matrices/parse.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides functions to parse and validate matrix expressions."""
8
9   from __future__ import annotations
10
11  import re
12  from dataclasses import dataclass
13  from typing import List, Pattern, Set, Tuple
14
15  from lintrans.typing_ import MatrixParseList
16
17  _ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
18
19  NAIVE_CHARACTER_CLASS = r'[-+\sA-Z0-9.rot()^{}\[\];]'
20  """This is a RegEx character class that just holds all the valid characters for an expression.
21
22  See :func:`validate_matrix_expression` to actually validate matrix expressions.
23  """
24
25
26  class MatrixParseError(Exception):
27      """A simple exception to be raised when an error is found when parsing."""
28
29
30  def compile_naive_expression_pattern() -> Pattern[str]:
31      """Compile the single RegEx pattern that will match a valid matrix expression."""
32      digit_no_zero = '[123456789]'
33      digits = '\\d+'
34      integer_no_zero = digit_no_zero + '(' + digits + ')?'
35      real_number = f'({integer_no_zero}(\\.{digits})?|0\\.{digits})'
36
37      anonymous_matrix = r'\[(-?\d+(?:\.\d+)?) (-?\d+(?:\.\d+)?);(-?\d+(?:\.\d+)?) (-?\d+(?:\.\d+)?)\]'
38
39      index_content = f'(-?{integer_no_zero}|T)'
40      index = f'(\\^{{{index_content}}}|\\^{index_content})'
41      matrix_identifier = f'([A-Z]|rot\\(-?{real_number}\\)|{anonymous_matrix}|\\({NAIVE_CHARACTER_CLASS}+\\))'
42      matrix = '(' + real_number + '?' + matrix_identifier + index + '?)'
43      expression = f'^-?{matrix}+((\\+-?|-){matrix}+)*$'
44
45      return re.compile(expression)
46
47
48  # This is an expensive pattern to compile, so we compile it when this module is initialized
49  _naive_expression_pattern = compile_naive_expression_pattern()
50
51
52  def find_sub_expressions(expression: str) -> List[str]:
53      """Find all the sub-expressions in the given expression.
54
55      This function only goes one level deep, so may return strings like ``'A(BC)D'``.
56
57      :raises MatrixParseError: If there are unbalanced parentheses
58      """
59      sub_expressions: List[str] = []
60      string = ''
61      paren_depth = 0
62      pointer = 0
63
64      expression = strip_whitespace(expression)
65
```

```
66          while True:
67              char = expression[pointer]
68
69              if char == '(' and expression[pointer - 3:pointer] != 'rot':
70                  paren_depth += 1
71
72                  # This is a bit of a manual bodge, but it eliminates extraneous parens
73                  if paren_depth == 1:
74                      pointer += 1
75                      continue
76
77              elif char == ')' and re.match(f'{NAIVE_CHARACTER_CLASS}*?rot\\([-\\d.]+$', expression[:pointer]) is None:
78                  paren_depth -= 1
79
80              if paren_depth > 0:
81                  string += char
82
83              if paren_depth == 0 and string:
84                  sub_expressions.append(string)
85                  string = ''
86
87              pointer += 1
88
89              if pointer >= len(expression):
90                  break
91
92          if paren_depth != 0:
93              raise MatrixParseError('Unbalanced parentheses in expression')
94
95          return sub_expressions
96
97
98      def strip_whitespace(expression: str) -> str:
99          """Strip the whitespace from the given expression, preserving whitespace in anonymous matrices.
100
101         Whitespace in anonymous matrices is preserved such that there is exactly one space in the middle of each pair of
102         numbers, but no space after the semi-colon, like so: ``[1 -2;3.4 5]``.
103         """
104         # We replace the necessary whitespace with null bytes to preserve it
105         expression = re.sub(
106             r'\[\s*(-?\d+(?:\.\d+)?)\s+(-?\d+(?:\.\d+)?)\s*;\s*(-?\d+(?:\.\d+)?)\s+(-?\d+(?:\.\d+)?)\s*\]',
107             r'[\g<1> \g<2>;\g<3> \g<4>]'.replace(' ', '\x00'),
108             expression
109         )
110
111         expression = re.sub(r'\s', '', expression)
112         return re.sub('\x00', ' ', expression)
113
114
115     def validate_matrix_expression(expression: str) -> bool:
116         """Validate the given matrix expression.
117
118         This function simply checks the expression against the BNF schema documented in
119         :ref:`expression-syntax-docs`. It is not aware of which matrices are actually defined
120         in a wrapper. For an aware version of this function, use the
121         :meth:`~lintrans.matrices.wrapper.MatrixWrapper.is_valid_expression` method on
122         :class:`~lintrans.matrices.wrapper.MatrixWrapper`.
123
124         :param str expression: The expression to be validated
125         :returns bool: Whether the expression is valid according to the schema
126         """
127         # Remove all whitespace
128         expression = strip_whitespace(expression)
129         match = _naive_expression_pattern.match(expression)
130
131         if match is None:
132             return False
133
134         if re.search(r'\^-?\d*\.\d+', expression) is not None:
135             return False
136
137         # Check that the whole expression was matched against
138         if expression != match.group(0):
```

```
            return False

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

        if len(sub_expressions) == 0:
            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()
        [[('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 = strip_whitespace(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}")'
```

```python

    @property
    def _char(self) -> str:
        """Return the character 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 == '+':
                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

```

```
284             if self._char.isdigit() or self._char == '-':
285                 if self._current_token.multiplier != '' \
286                         or (self._current_token.multiplier == '' and self._current_token.identifier != ''):
287                     return False
288 
289                 self._parse_multiplier()
290 
291             elif self._char.isalpha() and self._char.isupper():
292                 if self._current_token.identifier != '':
293                     return False
294 
295                 self._current_token.identifier = self._char
296                 self._pointer += 1
297 
298             elif self._char == 'r':
299                 if self._current_token.identifier != '':
300                     return False
301 
302                 self._parse_rot_identifier()
303 
304             elif self._char == '[':
305                 if self._current_token.identifier != '':
306                     return False
307 
308                 self._parse_anonymous_identifer()
309 
310             elif self._char == '(':
311                 if self._current_token.identifier != '':
312                     return False
313 
314                 self._parse_sub_expression()
315 
316             elif self._char == '^':
317                 if self._current_token.exponent != '':
318                     return False
319 
320                 self._parse_exponent()
321 
322             elif self._char == '+':
323                 return False
324 
325             else:
326                 raise MatrixParseError(f'Unrecognised character "{self._char}" in matrix expression')
327 
328             return True
329 
330         def _parse_multiplier(self) -> None:
331             """Parse a multiplier from the expression and pointer.
332 
333             This method just parses a numerical multiplier, which can include
334             zero or one ``.`` character and optionally a ``-`` at the start.
335 
336             :raises MatrixParseError: If we fail to parse this part of the matrix
337             """
338             multiplier = ''
339 
340             while self._char.isdigit() or self._char in ('.', '-'):
341                 multiplier += self._char
342                 self._pointer += 1
343 
344             try:
345                 float(multiplier)
346             except ValueError as e:
347                 raise MatrixParseError(f'Invalid multiplier "{multiplier}"') from e
348 
349             self._current_token.multiplier = multiplier
350 
351         def _parse_rot_identifier(self) -> None:
352             """Parse a ``rot()``-style identifier from the expression and pointer.
353 
354             This method will just parse something like ``rot(12.5)``. The angle number must be a real number.
355 
356             :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_anonymous_identifer(self) -> None:
            """Parse an anonymous matrix, including the square brackets."""
            if match := re.match(
                r'^\[(-?\d+(?:\.\d+)?) (-?\d+(?:\.\d+)?);(-?\d+(?:\.\d+)?) (-?\d+(?:\.\d+)?)\]',
                self._expression[self._pointer:]
            ):
                for n in range(1, 4 + 1):
                    try:
                        float(match.group(n))
                    except ValueError as e:
                        raise MatrixParseError(f'Invalid matrix entry "{match.group(1)}" in anonymous matrix') from e

                self._current_token.identifier = match.group(0)
                self._pointer += len(match.group(0))
            else:
                raise MatrixParseError(
                    f'Invalid anonymous matrix "{self._expression[self._pointer : self._pointer + 15]}..."'
                )

        def _parse_sub_expression(self) -> None:
            """Parse a parenthesized sub-expression as the identifier.

            This method will also validate the expression in the parentheses.

            :raises MatrixParseError: If we fail to parse this part of the matrix
            """
            if self._char != '(':
                raise MatrixParseError('Sub-expression must start with "("')

            self._pointer += 1
            paren_depth = 1
            identifier = ''

            while paren_depth > 0:
                if self._char == '(':
                    paren_depth += 1
                elif self._char == ')':
                    paren_depth -= 1

                if paren_depth == 0:
                    self._pointer += 1
                    break

                identifier += self._char
                self._pointer += 1

            if not validate_matrix_expression(identifier):
                raise MatrixParseError(f'Invalid sub-expression identifier "{identifier}"')

            self._current_token.identifier = identifier

        def _parse_exponent(self) -> None:
            """Parse a matrix exponent from the expression and pointer.

            The exponent must be an integer or ``T`` for transpose.

            :raises MatrixParseError: If we fail to parse this part of the token
            """
```

```
        if match := re.match(r'\^\{(-?\d+|T)\}', self._expression[self._pointer:]):
            exponent = match.group(1)

            try:
                if exponent != 'T':
                    int(exponent)
            except ValueError as e:
                raise MatrixParseError(f'Invalid exponent "{match.group(1)}"') from e

            self._current_token.exponent = exponent
            self._pointer += len(match.group(0))
        else:
            raise MatrixParseError(
                f'Invalid exponent "{self._expression[self._pointer : self._pointer + 10]}..."'
            )


def parse_matrix_expression(expression: str) -> MatrixParseList:
    """Parse the matrix expression and return a :attr:`~lintrans.typing_.MatrixParseList`.

    :Example:

    >>> parse_matrix_expression('A')
    [[('', 'A', '')]]
    >>> parse_matrix_expression('-3M^2')
    [[('-3', 'M', '2')]]
    >>> parse_matrix_expression('1.2rot(12)^{3}2B^T')
    [[('1.2', 'rot(12)', '3'), ('2', 'B', 'T')]]
    >>> parse_matrix_expression('A^2 + 3B')
    [[('', 'A', '2')], [('3', 'B', '')]]
    >>> parse_matrix_expression('-3A^{-1}3B^T - 45M^2')
    [[('-3', 'A', '-1'), ('3', 'B', 'T')], [('-45', 'M', '2')]]
    >>> parse_matrix_expression('5.3A^{4} 2.6B^{-2} + 4.6D^T 8.9E^{-1}')
    [[('5.3', 'A', '4'), ('2.6', 'B', '-2')], [('4.6', 'D', 'T'), ('8.9', 'E', '-1')]]
    >>> parse_matrix_expression('2(A+B^TC)^2D')
    [[('2', 'A+B^{T}C', '2'), ('', 'D', '')]]

    :param str expression: The expression to be parsed
    :returns: A list of parsed components
    :rtype: :attr:`~lintrans.typing_.MatrixParseList`
    """
    return ExpressionParser(expression).parse()


def get_matrix_identifiers(expression: str) -> Set[str]:
    """Return all the matrix identifiers used in the given expression.

    This method works recursively with sub-expressions.
    """
    s = set()
    top_level = [id for sublist in parse_matrix_expression(expression) for _, id, _ in sublist]

    for body in top_level:
        if body in _ALPHABET:
            s.add(body)

        elif re.match(r'rot\(\d+(\.\d+)?\)', body):
            continue

        else:
            s.update(get_matrix_identifiers(body))

    return s
```

A.10 `matrices/__init__.py`

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

# This program is licensed under GNU GPLv3, available here:
```

```
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This package supplies classes and functions to parse, evaluate, and wrap matrices."""
8
9   from . import parse, utility
10  from .utility import create_rotation_matrix
11  from .wrapper import MatrixWrapper
12
13  __all__ = ['create_rotation_matrix', 'MatrixWrapper', 'parse', 'utility']
```

A.11 gui/utility.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides utility functions for the whole GUI, such as :func:`qapp`."""
8
9   from PyQt5.QtCore import QCoreApplication
10
11
12  def qapp() -> QCoreApplication:
13      """Return the equivalent of the global :class:`qApp` pointer.
14
15      :raises RuntimeError: If :meth:`QCoreApplication.instance` returns ``None``
16      """
17      instance = QCoreApplication.instance()
18
19      if instance is None:
20          raise RuntimeError('qApp undefined')
21
22      return instance
```

A.12 gui/main_window.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides the :class:`LintransMainWindow` class, which provides the main window for the GUI."""
8
9   from __future__ import annotations
10
11  import os
12  import re
13  import sys
14  import webbrowser
15  from copy import deepcopy
16  from pathlib import Path
17  from pickle import UnpicklingError
18  from typing import List, NoReturn, Optional, Type
19
20  import numpy as np
21  from numpy import linalg
22  from numpy.linalg import LinAlgError
23  from PyQt5 import QtWidgets
24  from PyQt5.QtCore import QObject, Qt, QThread, pyqtSignal, pyqtSlot
25  from PyQt5.QtGui import QCloseEvent, QIcon, QKeyEvent, QKeySequence
26  from PyQt5.QtWidgets import (QAction, QApplication, QFileDialog, QHBoxLayout,
27                               QMainWindow, QMenu, QMessageBox, QPushButton,
28                               QShortcut, QSizePolicy, QSpacerItem,
29                               QStyleFactory, QVBoxLayout)
30
31  import lintrans
```

```
32      from lintrans import updating
33      from lintrans.global_settings import GlobalSettings, UpdateType
34      from lintrans.gui.dialogs.settings import GlobalSettingsDialog
35      from lintrans.matrices import MatrixWrapper
36      from lintrans.matrices.parse import validate_matrix_expression
37      from lintrans.matrices.utility import polar_coords, rotate_coord
38      from lintrans.typing_ import MatrixType, VectorType
39
40      from .dialogs import (AboutDialog, DefineAsExpressionDialog,
41                            DefineMatrixDialog, DefineNumericallyDialog,
42                            DefinePolygonDialog, DefineVisuallyDialog,
43                            DisplaySettingsDialog, FileSelectDialog, InfoPanelDialog,
44                            PromptUpdateDialog)
45      from .plots import MainViewportWidget
46      from .session import Session
47      from .settings import DisplaySettings
48      from .utility import qapp
49      from .validate import MatrixExpressionValidator
50
51
52      class _UpdateChecker(QObject):
53          """A simple class to act as a worker for a :class:`QThread`."""
54
55          signal_prompt_update: pyqtSignal = pyqtSignal(str)
56          """A signal that is emitted if a new version is found. The argument is the new version string."""
57
58          finished: pyqtSignal = pyqtSignal()
59          """A signal that is emitted when the worker has finished. Intended to be used for cleanup."""
60
61          def check_for_updates_and_emit(self) -> None:
62              """Check for updates, and emit :attr:`signal_prompt_update` if there's a new version.
63
64              This method exists to be run in a background thread to trigger a prompt if a new version is found.
65              """
66              update_type = GlobalSettings().get_data().update_type
67
68              if update_type == UpdateType.never:
69                  return
70
71              if update_type == UpdateType.auto:
72                  updating.update_lintrans_in_background(check=True)
73                  return
74
75              # If we get here, then update_type must be prompt,
76              # so we can check for updates and possibly prompt the user
77              new, version = updating.new_version_exists()
78              if new:
79                  self.signal_prompt_update.emit(version)
80
81              self.finished.emit()
82
83
84      class LintransMainWindow(QMainWindow):
85          """This class provides a main window for the GUI using the Qt framework.
86
87          This class should not be used directly, instead call :func:`main` to create the GUI.
88          """
89
90          def __init__(self):
91              """Create the main window object, and create and arrange every widget in it.
92
93              This doesn't show the window, it just constructs it. Use :func:`main` to show the GUI.
94              """
95              super().__init__()
96
97              self._matrix_wrapper = MatrixWrapper()
98
99              self._expression_history: List[str] = []
100             self._expression_history_index: Optional[int] = None
101
102             self.setWindowTitle('[*]lintrans')
103             self.setMinimumSize(800, 650)
104
```

```
105             path = Path(__file__).parent.absolute() / 'assets' / 'icon.jpg'
106             self.setWindowIcon(QIcon(str(path)))
107 
108             self._animating: bool = False
109             self._animating_sequence: bool = False
110             self._reset_during_animation: bool = False
111 
112             self._save_filename: Optional[str] = None
113 
114             # Set up thread and worker to check for updates
115 
116             self._thread_updates = QThread()
117             self._worker_updates = _UpdateChecker()
118             self._worker_updates.moveToThread(self._thread_updates)
119 
120             self._thread_updates.started.connect(self._worker_updates.check_for_updates_and_emit)
121             self._worker_updates.signal_prompt_update.connect(self._prompt_update)
122             self._worker_updates.finished.connect(self._thread_updates.quit)
123             self._worker_updates.finished.connect(self._worker_updates.deleteLater)
124             self._thread_updates.finished.connect(self._thread_updates.deleteLater)
125 
126             # === Create menubar
127 
128             menubar = QtWidgets.QMenuBar(self)
129 
130             menu_file = QMenu(menubar)
131             menu_file.setTitle('&File')
132 
133             menu_help = QMenu(menubar)
134             menu_help.setTitle('&Help')
135 
136             action_global_settings = QAction(self)
137             action_global_settings.setText('Settings')
138             action_global_settings.setShortcut('Ctrl+Alt+S')
139             action_global_settings.triggered.connect(self._dialog_change_global_settings)
140 
141             action_reset_session = QAction(self)
142             action_reset_session.setText('Reset session')
143             action_reset_session.triggered.connect(self._reset_session)
144 
145             action_open = QAction(self)
146             action_open.setText('&Open')
147             action_open.setShortcut('Ctrl+O')
148             action_open.triggered.connect(self._ask_for_session_file)
149 
150             action_save = QAction(self)
151             action_save.setText('&Save')
152             action_save.setShortcut('Ctrl+S')
153             action_save.triggered.connect(self._save_session)
154 
155             action_save_as = QAction(self)
156             action_save_as.setText('Save as...')
157             action_save_as.setShortcut('Ctrl+Shift+S')
158             action_save_as.triggered.connect(self._save_session_as)
159 
160             action_quit = QAction(self)
161             action_quit.setText('&Quit')
162             action_quit.triggered.connect(self.close)
163 
164             # If this is an old release, use the docs for this release. Else, use the latest docs
165             # We use the latest because most use cases for non-stable releases will be in development and testing
166             docs_link = 'https://lintrans.readthedocs.io/en/'
167 
168             if re.match(r'^\d+\.\d+\.\d+$', lintrans.__version__):
169                 docs_link += 'v' + lintrans.__version__
170             else:
171                 docs_link += 'latest'
172 
173             action_tutorial = QAction(self)
174             action_tutorial.setText('&Tutorial')
175             action_tutorial.setShortcut('F1')
176             action_tutorial.triggered.connect(
177                 lambda: webbrowser.open_new_tab(docs_link + '/tutorial/index.html')
```

```
178             )
179
180             action_docs = QAction(self)
181             action_docs.setText('&Docs')
182             action_docs.triggered.connect(
183                 lambda: webbrowser.open_new_tab(docs_link + '/backend/lintrans.html')
184             )
185
186             menu_feedback = QMenu(menu_help)
187             menu_feedback.setTitle('Give feedback')
188
189             action_bug_report = QAction(self)
190             action_bug_report.setText('Report a bug')
191             action_bug_report.triggered.connect(
192                 lambda: webbrowser.open_new_tab('https://forms.gle/Q82cLTtgPLcV4xQD6')
193             )
194
195             action_suggest_feature = QAction(self)
196             action_suggest_feature.setText('Suggest a new feature')
197             action_suggest_feature.triggered.connect(
198                 lambda: webbrowser.open_new_tab('https://forms.gle/mVWbHiMBw9Zq5Ze37')
199             )
200
201             menu_feedback.addAction(action_bug_report)
202             menu_feedback.addAction(action_suggest_feature)
203
204             action_about = QAction(self)
205             action_about.setText('&About')
206             action_about.triggered.connect(lambda: AboutDialog(self).open())
207
208             menu_file.addAction(action_global_settings)
209             menu_file.addSeparator()
210             menu_file.addAction(action_reset_session)
211             menu_file.addAction(action_open)
212             menu_file.addSeparator()
213             menu_file.addAction(action_save)
214             menu_file.addAction(action_save_as)
215             menu_file.addSeparator()
216             menu_file.addAction(action_quit)
217
218             menu_help.addAction(action_tutorial)
219             menu_help.addAction(action_docs)
220             menu_help.addSeparator()
221             menu_help.addMenu(menu_feedback)
222             menu_help.addSeparator()
223             menu_help.addAction(action_about)
224
225             menubar.addAction(menu_file.menuAction())
226             menubar.addAction(menu_help.menuAction())
227
228             self.setMenuBar(menubar)
229
230             # === Create widgets
231
232             # Left layout: the plot and input box
233
234             self._plot = MainViewportWidget(
235                 self,
236                 display_settings=GlobalSettings().get_display_settings(),
237                 polygon_points=[]
238             )
239
240             self._lineedit_expression_box = QtWidgets.QLineEdit(self)
241             self._lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
242             self._lineedit_expression_box.setValidator(MatrixExpressionValidator(self))
243             self._lineedit_expression_box.textChanged.connect(self._update_render_buttons)
244
245             # Right layout: all the buttons
246
247             # Misc buttons
248
249             button_define_polygon = QPushButton(self)
250             button_define_polygon.setText('Define polygon')
```

```
251             button_define_polygon.clicked.connect(self._dialog_define_polygon)
252             button_define_polygon.setToolTip('Define a polygon to view its transformation<br><b>(Ctrl + P)</b>')
253             QShortcut(QKeySequence('Ctrl+P'), self).activated.connect(button_define_polygon.click)
254 
255             self._button_change_display_settings = QPushButton(self)
256             self._button_change_display_settings.setText('Change\ndisplay settings')
257             self._button_change_display_settings.clicked.connect(self._dialog_change_display_settings)
258             self._button_change_display_settings.setToolTip(
259                 "Change which things are rendered and how they're rendered<br><b>(Ctrl + D)</b>"
260             )
261             QShortcut(QKeySequence('Ctrl+D'), self).activated.connect(self._button_change_display_settings.click)
262 
263             button_reset_zoom = QPushButton(self)
264             button_reset_zoom.setText('Reset zoom')
265             button_reset_zoom.clicked.connect(self._reset_zoom)
266             button_reset_zoom.setToolTip('Reset the zoom level back to normal<br><b>(Ctrl + Shift + R)</b>')
267             QShortcut(QKeySequence('Ctrl+Shift+R'), self).activated.connect(button_reset_zoom.click)
268 
269             # Define new matrix buttons and their groupbox
270 
271             self._button_define_visually = QPushButton(self)
272             self._button_define_visually.setText('Visually')
273             self._button_define_visually.setToolTip('Drag the basis vectors<br><b>(Alt + 1)</b>')
274             self._button_define_visually.clicked.connect(lambda: self._dialog_define_matrix(DefineVisuallyDialog))
275             QShortcut(QKeySequence('Alt+1'), self).activated.connect(self._button_define_visually.click)
276 
277             self._button_define_numerically = QPushButton(self)
278             self._button_define_numerically.setText('Numerically')
279             self._button_define_numerically.setToolTip('Define a matrix just with numbers<br><b>(Alt + 2)</b>')
280             self._button_define_numerically.clicked.connect(lambda: self._dialog_define_matrix(DefineNumericallyDialog))
281             QShortcut(QKeySequence('Alt+2'), self).activated.connect(self._button_define_numerically.click)
282 
283             self._button_define_as_expression = QPushButton(self)
284             self._button_define_as_expression.setText('As an expression')
285             self._button_define_as_expression.setToolTip('Define a matrix in terms of other matrices<br><b>(Alt +
                ↪  3)</b>')
286             self._button_define_as_expression.clicked.connect(
287                 lambda: self._dialog_define_matrix(DefineAsExpressionDialog)
288             )
289             QShortcut(QKeySequence('Alt+3'), self).activated.connect(self._button_define_as_expression.click)
290 
291             vlay_define_new_matrix = QVBoxLayout()
292             vlay_define_new_matrix.setSpacing(20)
293             vlay_define_new_matrix.addWidget(self._button_define_visually)
294             vlay_define_new_matrix.addWidget(self._button_define_numerically)
295             vlay_define_new_matrix.addWidget(self._button_define_as_expression)
296 
297             groupbox_define_new_matrix = QtWidgets.QGroupBox('Define a new matrix', self)
298             groupbox_define_new_matrix.setLayout(vlay_define_new_matrix)
299 
300             # Info panel button
301 
302             self._button_info_panel = QPushButton(self)
303             self._button_info_panel.setText('Show defined matrices')
304             self._button_info_panel.clicked.connect(self._open_info_panel)
305             self._button_info_panel.setToolTip(
306                 'Open an info panel with all matrices that have been defined in this session<br><b>(Ctrl + M)</b>'
307             )
308             QShortcut(QKeySequence('Ctrl+M'), self).activated.connect(self._button_info_panel.click)
309 
310             # Render buttons
311 
312             button_reset = QPushButton(self)
313             button_reset.setText('Reset')
314             button_reset.clicked.connect(self._reset_transformation)
315             button_reset.setToolTip('Reset the visualized transformation back to the identity<br><b>(Ctrl + R)</b>')
316             QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(button_reset.click)
317 
318             self._button_render = QPushButton(self)
319             self._button_render.setText('Render')
320             self._button_render.setEnabled(False)
321             self._button_render.clicked.connect(self._render_expression)
322             self._button_render.setToolTip('Render the expression<br><b>(Ctrl + Enter)</b>')
```

```
323             QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self._button_render.click)
324
325             self._button_animate = QPushButton(self)
326             self._button_animate.setText('Animate')
327             self._button_animate.setEnabled(False)
328             self._button_animate.clicked.connect(self._animate_expression)
329             self._button_animate.setToolTip('Animate the expression<br><b>(Ctrl + Shift + Enter)</b>')
330             QShortcut(QKeySequence('Ctrl+Shift+Return'), self).activated.connect(self._button_animate.click)
331
332             # === Arrange widgets
333
334             vlay_left = QVBoxLayout()
335             vlay_left.addWidget(self._plot)
336             vlay_left.addWidget(self._lineedit_expression_box)
337
338             vlay_misc_buttons = QVBoxLayout()
339             vlay_misc_buttons.setSpacing(20)
340             vlay_misc_buttons.addWidget(button_define_polygon)
341             vlay_misc_buttons.addWidget(self._button_change_display_settings)
342             vlay_misc_buttons.addWidget(button_reset_zoom)
343
344             vlay_info_buttons = QVBoxLayout()
345             vlay_info_buttons.setSpacing(20)
346             vlay_info_buttons.addWidget(self._button_info_panel)
347
348             vlay_render = QVBoxLayout()
349             vlay_render.setSpacing(20)
350             vlay_render.addWidget(button_reset)
351             vlay_render.addWidget(self._button_animate)
352             vlay_render.addWidget(self._button_render)
353
354             vlay_right = QVBoxLayout()
355             vlay_right.setSpacing(50)
356             vlay_right.addLayout(vlay_misc_buttons)
357             vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
358             vlay_right.addWidget(groupbox_define_new_matrix)
359             vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
360             vlay_right.addLayout(vlay_info_buttons)
361             vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
362             vlay_right.addLayout(vlay_render)
363
364             hlay_all = QHBoxLayout()
365             hlay_all.setSpacing(15)
366             hlay_all.addLayout(vlay_left)
367             hlay_all.addLayout(vlay_right)
368
369             central_widget = QtWidgets.QWidget()
370             central_widget.setLayout(hlay_all)
371             central_widget.setContentsMargins(10, 10, 10, 10)
372
373             self.setCentralWidget(central_widget)
374
375         def closeEvent(self, event: QCloseEvent) -> None:
376             """Handle a :class:`QCloseEvent` by confirming if the user wants to save, and cancelling animation."""
377             if not self.isWindowModified():
378                 self._animating = False
379                 self._animating_sequence = False
380                 GlobalSettings().save_display_settings(self._plot.display_settings)
381                 event.accept()
382                 return
383
384             if self._save_filename is not None:
385                 text = f"If you don't save, then changes made to {self._save_filename} will be lost."
386             else:
387                 text = "If you don't save, then changes made will be lost."
388
389             dialog = QMessageBox(self)
390             dialog.setIcon(QMessageBox.Question)
391             dialog.setWindowTitle('Save changes?')
392             dialog.setText(text)
393             dialog.setStandardButtons(QMessageBox.Save | QMessageBox.Discard | QMessageBox.Cancel)
394             dialog.setDefaultButton(QMessageBox.Save)
395
```

```
396             pressed_button = dialog.exec()
397
398             if pressed_button == QMessageBox.Save:
399                 self._save_session()
400
401             if pressed_button in (QMessageBox.Save, QMessageBox.Discard):
402                 self._animating = False
403                 self._animating_sequence = False
404                 GlobalSettings().save_display_settings(self._plot.display_settings)
405                 event.accept()
406             else:
407                 event.ignore()
408
409         def keyPressEvent(self, event: QKeyEvent) -> None:
410             """Handle a :class:`QKeyEvent` by scrolling through expression history."""
411             key = event.key()
412
413             # Load previous expression
414             if key == Qt.Key_Up:
415                 if self._expression_history_index is None:
416                     if len(self._expression_history) == 0:
417                         event.ignore()
418                         return
419
420                     # If the index is none and we've got a history, set the index to -1
421                     self._expression_history_index = -1
422
423                 # If the index is in range of the list (the index is always negative), then decrement it
424                 elif self._expression_history_index > -len(self._expression_history):
425                     self._expression_history_index -= 1
426
427                 self._lineedit_expression_box.setText(self._expression_history[self._expression_history_index])
428
429             # Load next expression
430             elif key == Qt.Key_Down:
431                 if self._expression_history_index is None:
432                     event.ignore()
433                     return
434
435                 self._expression_history_index += 1
436
437                 # The index is always negative, so if we've reached 0, then we need to stop
438                 if self._expression_history_index == 0:
439                     self._expression_history_index = None
440                     self._lineedit_expression_box.setText('')
441                 else:
442                     self._lineedit_expression_box.setText(self._expression_history[self._expression_history_index])
443
444             else:
445                 event.ignore()
446                 return
447
448             event.accept()
449
450         def _update_render_buttons(self) -> None:
451             """Enable or disable the render and animate buttons according to whether the matrix expression is valid."""
452             text = self._lineedit_expression_box.text()
453
454             # Let's say that the user defines a non-singular matrix A, then defines B as A^-1
455             # If they then redefine A and make it singular, then we get a LinAlgError when
456             # trying to evaluate an expression with B in it
457             # To fix this, we just do naive validation rather than aware validation
458             if ',' in text:
459                 self._button_render.setEnabled(False)
460
461                 try:
462                     valid = all(self._matrix_wrapper.is_valid_expression(x) for x in text.split(','))
463                 except LinAlgError:
464                     valid = all(validate_matrix_expression(x) for x in text.split(','))
465
466                 self._button_animate.setEnabled(valid)
467
468             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)

    def _extend_expression_history(self, text: str) -> None:
        """Extend the expression history with the given expression."""
        if len(self._expression_history) == 0 or self._expression_history[-1] != text:
            self._expression_history.append(text)
            self._expression_history_index = -1

    @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."""
        if self._animating or self._animating_sequence:
            self._reset_during_animation = True

        self._animating = False
        self._animating_sequence = False

        self._plot.plot_matrix(self._matrix_wrapper['I'])
        self._plot.update()

    @pyqtSlot()
    def _render_expression(self) -> None:
        """Render the transformation given by the expression in the input box."""
        try:
            text = self._lineedit_expression_box.text()
            matrix = self._matrix_wrapper.evaluate_expression(text)

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

        self._extend_expression_history(text)

        if self._is_matrix_too_big(matrix):
            return

        self._plot.plot_matrix(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()

        self._extend_expression_history(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]:
                if not self._animating_sequence:
                    break

                try:
                    new_matrix = self._matrix_wrapper.evaluate_expression(expr)

                    if self._plot.display_settings.applicative_animation:
                        new_matrix = new_matrix @ current_matrix

                    # 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(current_matrix - new_matrix) < 1e-12).all():
                        current_matrix = self._matrix_wrapper['I']

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

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

                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 its 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

            # 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.plot_matrix(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, the same length, and at right angles
        # 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)

        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
```

```
688             # This just scales the determinant along with the animation
689
690             # That is all of course, if we can do that
691             # We'll crash if we try to do this with det(B) == 0
692             if det_b == 0:
693                 return matrix_a
694
695             scalar: float = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
696             return scalar * matrix_b
697
698         def _animate_between_matrices(self, matrix_start: MatrixType, matrix_target: MatrixType) -> None:
699             """Animate from the start matrix to the target matrix."""
700             self._animating = True
701
702             # Making steps depend on animation_time ensures a smooth animation without
703             # massive overheads for small animation times
704             steps = self._plot.display_settings.animation_time // 10
705
706             for i in range(0, steps + 1):
707                 if not self._animating:
708                     break
709
710                 matrix_to_render = self._get_animation_frame(matrix_start, matrix_target, i / steps)
711
712                 if self._is_matrix_too_big(matrix_to_render):
713                     self._animating = False
714                     self._animating_sequence = False
715                     return
716
717                 self._plot.plot_matrix(matrix_to_render)
718
719                 # We schedule the plot to be updated, tell the event loop to
720                 # process events, and asynchronously sleep for 10ms
721                 # This allows for other events to be processed while animating, like zooming in and out
722                 self._plot.update()
723                 QApplication.processEvents()
724                 QThread.msleep(self._plot.display_settings.animation_time // steps)
725
726             if not self._reset_during_animation:
727                 self._plot.plot_matrix(matrix_target)
728             else:
729                 self._plot.plot_matrix(self._matrix_wrapper['I'])
730
731             self._plot.update()
732
733             self._animating = False
734             self._reset_during_animation = False
735
736         @pyqtSlot()
737         def _open_info_panel(self) -> None:
738             """Open the info panel and register a callback to undefine matrices."""
739             dialog = InfoPanelDialog(self._matrix_wrapper, self)
740             dialog.open()
741             dialog.finished.connect(self._assign_matrix_wrapper)
742
743         @pyqtSlot(DefineMatrixDialog)
744         def _dialog_define_matrix(self, dialog_class: Type[DefineMatrixDialog]) -> None:
745             """Open a generic definition dialog to define a new matrix.
746
747             The class for the desired dialog is passed as an argument. We create an
748             instance of this class and the dialog is opened asynchronously and modally
749             (meaning it blocks interaction with the main window) with the proper method
750             connected to the :meth:`QDialog.accepted` signal.
751
752             .. note:: ``dialog_class`` must subclass
    ↪  :class:`~lintrans.gui.dialogs.define_new_matrix.DefineMatrixDialog`.
753
754             :param dialog_class: The dialog class to instantiate
755             :type dialog_class: Type[lintrans.gui.dialogs.define_new_matrix.DefineMatrixDialog]
756             """
757             # We create a dialog with a deepcopy of the current matrix_wrapper
758             # This avoids the dialog mutating this one
759             dialog: DefineMatrixDialog
```

```
760
761             if dialog_class == DefineVisuallyDialog:
762                 dialog = DefineVisuallyDialog(
763                     self,
764                     matrix_wrapper=deepcopy(self._matrix_wrapper),
765                     display_settings=self._plot.display_settings,
766                     polygon_points=self._plot.polygon_points,
767                     input_vector=self._plot.point_input_vector
768                 )
769             else:
770                 dialog = dialog_class(self, matrix_wrapper=deepcopy(self._matrix_wrapper))
771
772             # .open() is asynchronous and doesn't spawn a new event loop, but the dialog is still modal (blocking)
773             dialog.open()
774
775             # So we have to use the accepted signal to call a method when the user accepts the dialog
776             dialog.accepted.connect(self._assign_matrix_wrapper)
777
778         @pyqtSlot()
779         def _assign_matrix_wrapper(self) -> None:
780             """Assign a new value to ``self._matrix_wrapper`` and give the expression box focus."""
781             self._matrix_wrapper = self.sender().matrix_wrapper
782             self._lineedit_expression_box.setFocus()
783             self._update_render_buttons()
784
785             self.setWindowModified(True)
786             self._update_window_title()
787
788         @pyqtSlot()
789         def _dialog_change_global_settings(self) -> None:
790             """Open the dialog to change the global settings."""
791             dialog = GlobalSettingsDialog(self)
792             dialog.open()
793             dialog.accepted.connect(self._plot.update)
794
795         @pyqtSlot()
796         def _dialog_change_display_settings(self) -> None:
797             """Open the dialog to change the display settings."""
798             dialog = DisplaySettingsDialog(self, display_settings=self._plot.display_settings)
799             dialog.open()
800             dialog.accepted.connect(self._assign_display_settings)
801
802         @pyqtSlot()
803         def _assign_display_settings(self) -> None:
804             """Assign a new value to ``self._plot.display_settings`` and give the expression box focus."""
805             self._plot.display_settings = self.sender().display_settings
806             self._plot.update()
807             self._lineedit_expression_box.setFocus()
808             self._update_render_buttons()
809
810         @pyqtSlot()
811         def _dialog_define_polygon(self) -> None:
812             """Open the dialog to define a polygon."""
813             dialog = DefinePolygonDialog(self, polygon_points=self._plot.polygon_points)
814             dialog.open()
815             dialog.accepted.connect(self._assign_polygon_points)
816
817         @pyqtSlot()
818         def _assign_polygon_points(self) -> None:
819             """Assign a new value to ``self._plot.polygon_points`` and give the expression box focus."""
820             self._plot.polygon_points = self.sender().polygon_points
821             self._plot.update()
822             self._lineedit_expression_box.setFocus()
823             self._update_render_buttons()
824
825             self.setWindowModified(True)
826             self._update_window_title()
827
828         def _show_error_message(self, title: str, text: str, info: str | None = None, *, warning: bool = False) -> None:
829             """Show an error message in a dialog box.
830
831             :param str title: The window title of the dialog box
832             :param str text: The simple error message
```

```
            :param info: The more informative error message
            :type info: Optional[str]
            """
            dialog = QMessageBox(self)
            dialog.setWindowTitle(title)
            dialog.setText(text)

            if warning:
                dialog.setIcon(QMessageBox.Warning)
            else:
                dialog.setIcon(QMessageBox.Critical)

            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 on the grid.

            We're checking against a 1000x1000 grid here, which is far less than the actual space we have available.
            But even when fully zoomed out 1080p monitor, the grid is only roughly 170x90, so 1000x1000 is plenty.

            :param MatrixType matrix: The matrix to check
            :returns bool: Whether the matrix is too big to fit on the canvas
            """
            for x, y in matrix.T:
                if not (-1000 <= x <= 1000 and -1000 <= y <= 1000):
                    self._show_error_message(
                        'Matrix too big',
                        "This matrix doesn't fit on the grid.",
                        'This grid is only 1000x1000, and this matrix\n'
                        f'[{int(matrix[0][0])} {int(matrix[0][1])}; {int(matrix[1][0])} {int(matrix[1][1])}]\n'
                        " doesn't fit."
                    )
                    return True

            return False

        def _update_window_title(self) -> None:
            """Update the window title to reflect whether the session has changed since it was last saved."""
            if self._save_filename:
                title = os.path.split(self._save_filename)[-1] + '[*] - lintrans'
            else:
                title = '[*]lintrans'

            self.setWindowTitle(title)

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

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

            if dialog.exec() == QMessageBox.Yes:
                self._matrix_wrapper = MatrixWrapper()
                self._plot.polygon_points = []
                self._plot.display_settings = GlobalSettings().get_display_settings()

                self._reset_transformation()
                self._expression_history = []
                self._expression_history_index = None
                self._lineedit_expression_box.setText('')
                self._lineedit_expression_box.setFocus()
```

```
            self._update_render_buttons()

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

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

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

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

        missing_parts = False

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

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

        if session.display_settings is not None:
            self._plot.display_settings = session.display_settings
        else:
            self._plot.display_settings = DisplaySettings()  # type: ignore[unreachable]
            missing_parts = True

        if session.input_vector is not None:
            self._plot.point_input_vector = session.input_vector
        else:
            self._plot.point_input_vector = (1, 1)  # type: ignore[unreachable]
            missing_parts = True

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

            self._show_error_message(
                'Session file missing parts',
                'This session file is missing certain elements. It may not work correctly.',
                info,
                warning=True
            )
        elif extra_attrs:
            if version != lintrans.__version__:
                info = f"This may be a version conflict. This file was saved with lintrans v{version} " \
                       f"but you're running lintrans v{lintrans.__version__}."
            else:
                info = None

            self._show_error_message(
```

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

        self._reset_transformation()
        self._expression_history = []
        self._expression_history_index = None
        self._lineedit_expression_box.setText('')
        self._lineedit_expression_box.setFocus()
        self._update_render_buttons()

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

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

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

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

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

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

        self.setWindowModified(False)
        self._update_window_title()

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

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

```
1052
1053            if dialog.exec():
1054                filename = dialog.selectedFiles()[0]
1055                self._save_filename = filename
1056                self._save_session()
1057
1058        @pyqtSlot(str)
1059        def _prompt_update(self, version: str) -> None:
1060            """Open a modal dialog to prompt the user to update lintrans."""
1061            dialog = PromptUpdateDialog(self, new_version=version)
1062            dialog.open()
1063
1064        def check_for_updates_and_prompt(self) -> None:
1065            """Update lintrans depending on the user's choice of update type.
1066
1067            If they chose 'prompt', then this method will open a prompt dialog (after checking
1068            if a new version actually exists). See :meth:`_prompt_update`.
1069            """
1070            self._thread_updates.start()
1071
1072
1073    def main(filename: Optional[str]) -> NoReturn:
1074        """Run the GUI by creating and showing an instance of :class:`LintransMainWindow`.
1075
1076        :param Optional[str] filename: A session file to optionally open at startup
1077        """
1078        app = QApplication([])
1079        app.setApplicationName('lintrans')
1080        app.setApplicationVersion(lintrans.__version__)
1081
1082        qapp().setStyle(QStyleFactory.create('fusion'))
1083
1084        window = LintransMainWindow()
1085        window.show()
1086        window.check_for_updates_and_prompt()
1087
1088        if filename:
1089            window.open_session_file(filename)
1090
1091        sys.exit(app.exec_())
```

A.13 gui/settings.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module contains the :class:`DisplaySettings` class, which holds configuration for display."""
8
9   from __future__ import annotations
10
11  import os
12  import pathlib
13  import pickle
14  from dataclasses import dataclass
15  from typing import Tuple
16
17  import lintrans
18
19
20  @dataclass(slots=True)
21  class DisplaySettings:
22      """This class simply holds some attributes to configure display."""
23
24      # === Basic stuff
25
26      draw_background_grid: bool = True
27      """This controls whether we want to draw the background grid.
```

```
28
29          The background axes will always be drawn. This makes it easy to identify the center of the space.
30          """
31
32          draw_transformed_grid: bool = True
33          """This controls whether we want to draw the transformed grid. Vectors are handled separately."""
34
35          draw_basis_vectors: bool = True
36          """This controls whether we want to draw the transformed basis vectors."""
37
38          label_basis_vectors: bool = False
39          """This controls whether we want to label the `i` and `j` basis vectors."""
40
41          # === Animations
42
43          smoothen_determinant: bool = True
44          """This controls whether we want the determinant to change smoothly during the animation.
45
46          .. note::
47             Even if this is ``True``, it will be ignored if we're animating from a positive det matrix to
48             a negative det matrix, or vice versa, because if we try to smoothly animate that determinant,
49             things blow up and the app often crashes.
50          """
51
52          applicative_animation: bool = True
53          """There are two types of simple animation, transitional and applicative.
54
55          Let ``C`` be the matrix representing the currently displayed transformation, and let ``T`` be the target matrix.
56          Transitional animation means that we animate directly from ``C`` from ``T``,
57          and applicative animation means that we animate from ``C`` to ``TC``, so we apply ``T`` to ``C``.
58          """
59
60          animation_time: int = 1200
61          """This is the number of milliseconds that an animation takes."""
62
63          animation_pause_length: int = 400
64          """This is the number of milliseconds that we wait between animations when using comma syntax."""
65
66          # === Matrix info
67
68          draw_determinant_parallelogram: bool = False
69          """This controls whether or not we should shade the parallelogram representing the determinant of the matrix."""
70
71          show_determinant_value: bool = True
72          """This controls whether we should write the text value of the determinant inside the parallelogram.
73
74          The text only gets draw if :attr:`draw_determinant_parallelogram` is also True.
75          """
76
77          draw_eigenvectors: bool = False
78          """This controls whether we should draw the eigenvectors of the transformation."""
79
80          draw_eigenlines: bool = False
81          """This controls whether we should draw the eigenlines of the transformation."""
82
83          # === Polygon
84
85          draw_untransformed_polygon: bool = True
86          """This controls whether we should draw the untransformed version of the user-defined polygon."""
87
88          draw_transformed_polygon: bool = True
89          """This controls whether we should draw the transformed version of the user-defined polygon."""
90
91          # === Input/output vectors
92
93          draw_input_vector: bool = True
94          """This controls whether we should draw the input vector in the main viewport."""
95
96          draw_output_vector: bool = True
97          """This controls whether we should draw the output vector in the main viewport."""
98
99          def save_to_file(self, filename: str) -> None:
100             """Save the display settings to a file, creating parent directories as needed."""
```

```
101             parent_dir = pathlib.Path(os.path.expanduser(filename)).parent.absolute()
102
103             if not os.path.isdir(parent_dir):
104                 os.makedirs(parent_dir)
105
106             data: Tuple[str, DisplaySettings] = (lintrans.__version__, self)
107
108             with open(filename, 'wb') as f:
109                 pickle.dump(data, f, protocol=4)
110
111         @classmethod
112         def load_from_file(cls, filename: str) -> Tuple[str, DisplaySettings]:
113             """Return the display settings that were previously saved to ``filename`` along with some extra information.
114
115             The tuple we return has the version of lintrans that was used to save the file, and the data itself.
116
117             :raises EOFError: If the file doesn't contain a pickled Python object
118             :raises FileNotFoundError: If the file doesn't exist
119             :raises ValueError: If the file contains a pickled object of the wrong type
120             """
121             if not os.path.isfile(filename):
122                 return lintrans.__version__, cls()
123
124             with open(filename, 'rb') as f:
125                 file_data = pickle.load(f)
126
127             if not isinstance(file_data, tuple):
128                 raise ValueError(f'File {filename} contains pickled object of the wrong type (must be tuple)')
129
130             # Create a default object and overwrite the fields that we have
131             data = cls()
132             for attr in file_data[1].__slots__:
133                 # Try to get the attribute from the old data, but don't worry if we can't,
134                 # because that means it's from an older version, so we can use the default
135                 # values from `cls()`
136                 try:
137                     setattr(data, attr, getattr(file_data[1], attr))
138                 except AttributeError:
139                     pass
140
141             return file_data[0], data
```

A.14 gui/session.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides the :class:`Session` class, which provides a way to save and load sessions."""
8
9   from __future__ import annotations
10
11  import os
12  import pathlib
13  import pickle
14  from collections import defaultdict
15  from typing import Any, DefaultDict, List, Tuple
16
17  import lintrans
18  from lintrans.gui.settings import DisplaySettings
19  from lintrans.matrices import MatrixWrapper
20
21
22  def _return_none() -> None:
23      """Return None.
24
25      This function only exists to make the defaultdict in :class:`Session` pickle-able.
26      """
```

```
        return None


    class Session:
        """Hold information about a session and provide methods to save and load that data."""

        __slots__ = ('matrix_wrapper', 'polygon_points', 'display_settings', 'input_vector')
        matrix_wrapper: MatrixWrapper
        polygon_points: List[Tuple[float, float]]
        display_settings: DisplaySettings
        input_vector: Tuple[float, float]

        def __init__(
            self,
            *,
            matrix_wrapper: MatrixWrapper,
            polygon_points: List[Tuple[float, float]],
            display_settings: DisplaySettings,
            input_vector: Tuple[float, float],
        ) -> None:
            """Create a :class:`Session` object with the given data."""
            self.matrix_wrapper = matrix_wrapper
            self.polygon_points = polygon_points
            self.display_settings = display_settings
            self.input_vector = input_vector

        def save_to_file(self, filename: str) -> None:
            """Save the session state to a file, creating parent directories as needed."""
            parent_dir = pathlib.Path(os.path.expanduser(filename)).parent.absolute()

            if not os.path.isdir(parent_dir):
                os.makedirs(parent_dir)

            data_dict: DefaultDict[str, Any] = defaultdict(_return_none, lintrans=lintrans.__version__)
            for attr in self.__slots__:
                data_dict[attr] = getattr(self, attr)

            with open(filename, 'wb') as f:
                pickle.dump(data_dict, f, protocol=4)

        @classmethod
        def load_from_file(cls, filename: str) -> Tuple[Session, str, bool]:
            """Return the session state that was previously saved to ``filename`` along with some extra information.

            The tuple we return has the :class:`Session` object (with some possibly None arguments),
            the lintrans version that the file was saved under, and whether the file had any extra
            attributes that this version doesn't support.

            :raises AttributeError: For specific older versions of :class:`Session` before it used ``__slots__``
            :raises EOFError: If the file doesn't contain a pickled Python object
            :raises FileNotFoundError: If the file doesn't exist
            :raises ValueError: If the file contains a pickled object of the wrong type
            """
            with open(filename, 'rb') as f:
                data_dict = pickle.load(f)

            if not isinstance(data_dict, defaultdict):
                raise ValueError(f'File {filename} contains pickled object of the wrong type (must be defaultdict)')

            session = cls(
                matrix_wrapper=data_dict['matrix_wrapper'],
                polygon_points=data_dict['polygon_points'],
                display_settings=data_dict['display_settings'],
                input_vector=data_dict['input_vector'],
            )

            # Check if the file has more attributes than we expect
            # If it does, it's probably from a higher version of lintrans
            extra_attrs = len(
                set(data_dict.keys()).difference(
                    set(['lintrans', *cls.__slots__])
                )
            ) != 0
```

```

        return session, data_dict['lintrans'], extra_attrs
```

A.15 gui/__init__.py

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

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

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

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

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

A.16 gui/validate.py

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

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

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

from __future__ import annotations

import re
from typing import Tuple

from PyQt5.QtGui import QValidator

from lintrans.matrices import parse


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

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

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

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

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

A.17 gui/dialogs/misc.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 miscellaneous dialog classes like :class:`AboutDialog`."""

    from __future__ import annotations

    import os
    import platform
    from typing import Dict, List, Optional, Tuple, Union

    from PyQt5.QtCore import PYQT_VERSION_STR, QT_VERSION_STR, Qt, pyqtSlot
    from PyQt5.QtGui import QKeySequence
    from PyQt5.QtWidgets import (QDialog, QFileDialog, QGridLayout, QGroupBox,
                                 QHBoxLayout, QLabel, QPushButton, QRadioButton,
                                 QShortcut, QSizePolicy, QSpacerItem,
                                 QStackedLayout, QVBoxLayout, QWidget)

    import lintrans
    from lintrans.global_settings import GlobalSettings, UpdateType
    from lintrans.gui.plots import DefinePolygonWidget
    from lintrans.matrices import MatrixWrapper
    from lintrans.matrices.utility import round_float
    from lintrans.typing_ import MatrixType, is_matrix_type
    from lintrans.updating import update_lintrans_in_background


    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 __init__(self, *args, **kwargs) -> None:
            """Set the :cpp:enum:`Qt::WA_DeleteOnClose` attribute to ensure deletion of dialog."""
            super().__init__(*args, **kwargs)
            self.setAttribute(Qt.WA_DeleteOnClose)
            self.setWindowFlag(Qt.WindowContextHelpButtonHint, False)

        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.
        """

        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)
```

```
78
79              label_info = QLabel(self)
80              label_info.setText(
81                  'lintrans is a program designed to help visualise<br>'
82                  '2D linear transformations represented with matrices.<br><br>'
83                  "It's designed for teachers and students and all feedback<br>"
84                  'is greatly appreciated. Go to <em>Help</em> &gt; <em>Give feedback</em><br>'
85                  'to report a bug or suggest a new feature, or you can<br>email me directly at '
86                  '<a href="mailto:dyson.dyson@icloud.com" style="color: black;">dyson.dyson@icloud.com</a>.'
87              )
88              label_info.setAlignment(Qt.AlignCenter)
89              label_info.setTextFormat(Qt.RichText)
90              label_info.setOpenExternalLinks(True)
91
92              label_copyright = QLabel(self)
93              label_copyright.setText(
94                  'This program is free software.<br>Copyright 2021-2022 D. Dyson (DoctorDalek1963).<br>'
95                  'This program is licensed under GPLv3, which can be found '
96                  '<a href="https://www.gnu.org/licenses/gpl-3.0.html" style="color: black;">here</a>.'
97              )
98              label_copyright.setAlignment(Qt.AlignCenter)
99              label_copyright.setTextFormat(Qt.RichText)
100             label_copyright.setOpenExternalLinks(True)
101
102             # === Arrange the widgets
103
104             self.setContentsMargins(10, 10, 10, 10)
105
106             vlay = QVBoxLayout()
107             vlay.setSpacing(20)
108             vlay.addWidget(label_title)
109             vlay.addWidget(label_version_info)
110             vlay.addWidget(label_info)
111             vlay.addWidget(label_copyright)
112
113             self.setLayout(vlay)
114
115
116     class InfoPanelDialog(FixedSizeDialog):
117         """A simple dialog class to display an info panel that shows all currently defined matrices."""
118
119         def __init__(self, matrix_wrapper: MatrixWrapper, *args, **kwargs):
120             """Create the dialog box with all the widgets needed to show the information."""
121             super().__init__(*args, **kwargs)
122             self.matrix_wrapper = matrix_wrapper
123
124             self._matrices: Dict[str, Optional[Union[MatrixType, str]]] = {
125                 name: value
126                 for name, value in self.matrix_wrapper.get_defined_matrices()
127             }
128
129             self.setWindowTitle('Defined matrices')
130             self.setContentsMargins(10, 10, 10, 10)
131
132             self._stacked_layout = QStackedLayout(self)
133             self.setLayout(self._stacked_layout)
134
135             self._draw_ui()
136
137         def _draw_ui(self) -> None:
138             grid_layout = QGridLayout()
139             grid_layout.setSpacing(20)
140
141             for i, (name, value) in enumerate(self._matrices.items()):
142                 if value is None:
143                     continue
144
145                 grid_layout.addWidget(
146                     self._get_full_matrix_widget(name, value),
147                     i % 4,
148                     i // 4,
149                     Qt.AlignCenter
150                 )
```

```
151
152             container = QWidget(self)
153             container.setLayout(grid_layout)
154             self._stacked_layout.setCurrentIndex(self._stacked_layout.addWidget(container))
155
156         def _undefine_matrix(self, name: str) -> None:
157             """Undefine the given matrix and redraw the dialog."""
158             for x in self.matrix_wrapper.undefine_matrix(name):
159                 self._matrices[x] = None
160
161             self._draw_ui()
162
163         def _get_full_matrix_widget(self, name: str, value: Union[MatrixType, str]) -> QWidget:
164             """Return a :class:`QWidget` containing the whole matrix widget composition.
165
166             Each defined matrix will get a widget group. Each group will be a label for the name,
167             a label for '=', and a container widget to either show the matrix numerically, or to
168             show the expression that it's defined as.
169
170             See :meth:`_get_matrix_data_widget`.
171             """
172             bold_font = self.font()
173             bold_font.setBold(True)
174
175             label_name = QLabel(self)
176             label_name.setText(name)
177             label_name.setFont(bold_font)
178
179             widget_matrix = self._get_matrix_data_widget(value)
180
181             hlay = QHBoxLayout()
182             hlay.setSpacing(10)
183             hlay.addWidget(label_name)
184             hlay.addWidget(QLabel('=', self))
185             hlay.addWidget(widget_matrix)
186
187             vlay = QVBoxLayout()
188             vlay.setSpacing(10)
189             vlay.addLayout(hlay)
190
191             if name != 'I':
192                 button_undefine = QPushButton(self)
193                 button_undefine.setText('Undefine')
194                 button_undefine.clicked.connect(lambda: self._undefine_matrix(name))
195
196                 vlay.addWidget(button_undefine)
197
198             groupbox = QGroupBox(self)
199             groupbox.setContentsMargins(10, 10, 10, 10)
200             groupbox.setLayout(vlay)
201
202             lay = QVBoxLayout()
203             lay.setSpacing(0)
204             lay.addWidget(groupbox)
205
206             container = QWidget(self)
207             container.setLayout(lay)
208
209             return container
210
211         def _get_matrix_data_widget(self, matrix: Union[MatrixType, str]) -> QWidget:
212             """Return a :class:`QWidget` containing the value of the matrix.
213
214             If the matrix is defined as an expression, it will be a simple :class:`QLabel`.
215             If the matrix is defined as a matrix, it will be a :class:`QWidget` container
216             with multiple :class:`QLabel` objects in it.
217             """
218             if isinstance(matrix, str):
219                 label = QLabel(self)
220                 label.setText(matrix)
221                 return label
222
223             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)
            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')


class FileSelectDialog(QFileDialog):
    """A subclass of :class:`QFileDialog` that fixes an issue with the default suffix on UNIX platforms."""

    def selectedFiles(self) -> List[str]:
        """Return a list of strings containing the absolute paths of the selected files in the dialog.

        There is an issue on UNIX platforms where a hidden directory will be recognised as a suffix.
        For example, ``/home/dyson/.lintrans/saves/test`` should have ``.lt`` appended, but
        ``.lintrans/saves/test`` gets recognised as the suffix, so the default suffix is not added.

        To fix this, we just look at the basename and see if it needs a suffix added. We do this for
        every name in the list, but there should be just one name, since this class is only intended
        to be used for saving files. We still return the full list of filenames.
        """
        selected_files: List[str] = []

        for filename in super().selectedFiles():
            # path will be the full path of the file, without the extension
            # This method understands hidden directories on UNIX platforms
            path, ext = os.path.splitext(filename)

            if ext == '':
                ext = '.' + self.defaultSuffix()

            selected_files.append(''.join((path, ext)))

        return selected_files
```

```
297     class DefinePolygonDialog(FixedSizeDialog):
298         """This dialog class allows the use to define a polygon with :class:`DefinePolygonWidget`."""
299
300         def __init__(self, *args, polygon_points: List[Tuple[float, float]], **kwargs) -> None:
301             """Create the dialog with the :class:`DefinePolygonWidget` widget."""
302             super().__init__(*args, **kwargs)
303
304             self.setWindowTitle('Define a polygon')
305             self.setMinimumSize(700, 550)
306
307             self.polygon_points = polygon_points
308
309             # === Create the widgets
310
311             self._polygon_widget = DefinePolygonWidget(polygon_points=polygon_points)
312
313             button_confirm = QPushButton(self)
314             button_confirm.setText('Confirm')
315             button_confirm.clicked.connect(self._confirm_polygon)
316             button_confirm.setToolTip('Confirm this polygon<br><b>(Ctrl + Enter)</b>')
317             QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(button_confirm.click)
318
319             button_cancel = QPushButton(self)
320             button_cancel.setText('Cancel')
321             button_cancel.clicked.connect(self.reject)
322             button_cancel.setToolTip('Discard this polygon<br><b>(Escape)</b>')
323
324             button_reset = QPushButton(self)
325             button_reset.setText('Reset polygon')
326             button_reset.clicked.connect(self._polygon_widget.reset_polygon)
327             button_reset.setToolTip('Remove all points of the polygon<br><b>(Ctrl + R)</b>')
328             QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(button_reset.click)
329
330             # === Arrange the widgets
331
332             self.setContentsMargins(10, 10, 10, 10)
333
334             hlay_buttons = QHBoxLayout()
335             hlay_buttons.setSpacing(20)
336             hlay_buttons.addWidget(button_reset)
337             hlay_buttons.addItem(QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum))
338             hlay_buttons.addWidget(button_cancel)
339             hlay_buttons.addWidget(button_confirm)
340
341             vlay = QVBoxLayout()
342             vlay.setSpacing(20)
343             vlay.addWidget(self._polygon_widget)
344             vlay.addLayout(hlay_buttons)
345
346             self.setLayout(vlay)
347
348         @pyqtSlot()
349         def _confirm_polygon(self) -> None:
350             """Confirm the polygon that the user has defined."""
351             self.polygon_points = self._polygon_widget.points
352             self.accept()
353
354
355     class PromptUpdateDialog(FixedSizeDialog):
356         """A simple dialog to ask the user if they want to upgrade their lintrans installation."""
357
358         def __init__(self, *args, new_version: str, **kwargs) -> None:
359             """Create the dialog with all its widgets."""
360             super().__init__(*args, **kwargs)
361
362             if new_version.startswith('v'):
363                 new_version = new_version[1:]
364
365             self.setWindowTitle('Update available')
366
367             # === Create the widgets
368
369             label_info = QLabel(self)
```

```
370             label_info.setText(
371                 'A new version of lintrans is available!\n'
372                 f'({lintrans.__version__} -> {new_version})\n\n'
373                 'Would you like to update now?'
374             )
375             label_info.setAlignment(Qt.AlignCenter)
376
377             label_explanation = QLabel(self)
378             label_explanation.setText(
379                 'The update will run silently in the background, so you can keep using lintrans uninterrupted.\n'
380                 'You can change your choice at any time in File > Settings.'
381             )
382             label_explanation.setAlignment(Qt.AlignCenter)
383
384             font = label_explanation.font()
385             font.setPointSize(int(0.9 * font.pointSize()))
386             font.setItalic(True)
387             label_explanation.setFont(font)
388
389             groupbox_radio_buttons = QGroupBox(self)
390
391             self._radio_button_auto = QRadioButton('Always update automatically', groupbox_radio_buttons)
392             self._radio_button_prompt = QRadioButton('Always ask to update', groupbox_radio_buttons)
393             self._radio_button_never = QRadioButton('Never update', groupbox_radio_buttons)
394
395             # If this prompt is even appearing, then the update type must be 'prompt'
396             self._radio_button_prompt.setChecked(True)
397
398             button_remind_me_later = QPushButton('Remind me later', self)
399             button_remind_me_later.clicked.connect(lambda: self._save_choice_and_update(False))
400             button_remind_me_later.setShortcut(Qt.Key_Escape)
401             button_remind_me_later.setFocus()
402
403             button_update_now = QPushButton('Update now', self)
404             button_update_now.clicked.connect(lambda: self._save_choice_and_update(True))
405
406             # === Arrange the widgets
407
408             self.setContentsMargins(10, 10, 10, 10)
409
410             hlay_buttons = QHBoxLayout()
411             hlay_buttons.setSpacing(20)
412             hlay_buttons.addWidget(button_remind_me_later)
413             hlay_buttons.addWidget(button_update_now)
414
415             vlay = QVBoxLayout()
416             vlay.setSpacing(20)
417             vlay.addWidget(label_info)
418
419             vlay_radio_buttons = QVBoxLayout()
420             vlay_radio_buttons.setSpacing(10)
421             vlay_radio_buttons.addWidget(self._radio_button_auto)
422             vlay_radio_buttons.addWidget(self._radio_button_prompt)
423             vlay_radio_buttons.addWidget(self._radio_button_never)
424
425             groupbox_radio_buttons.setLayout(vlay_radio_buttons)
426
427             vlay.addWidget(groupbox_radio_buttons)
428             vlay.addWidget(label_explanation)
429             vlay.addLayout(hlay_buttons)
430
431             self.setLayout(vlay)
432
433         def _save_choice_and_update(self, update_now: bool) -> None:
434             """Save the user's choice of how to update and optionally trigger an update now."""
435             gs = GlobalSettings()
436             if self._radio_button_auto.isChecked():
437                 gs.set_update_type(UpdateType.auto)
438
439             elif self._radio_button_prompt.isChecked():
440                 gs.set_update_type(UpdateType.prompt)
441
442             elif self._radio_button_never.isChecked():
```

```
443             gs.set_update_type(UpdateType.never)
444
445         if update_now:
446             # We don't need to check because we'll only get here if we know a new version is available
447             update_lintrans_in_background(check=False)
448             self.accept()
449         else:
450             self.reject()
```

A.18 gui/dialogs/settings.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides dialogs to edit settings within the app."""
8
9   from __future__ import annotations
10
11  import abc
12  from typing import Dict
13
14  from PyQt5 import QtWidgets
15  from PyQt5.QtCore import Qt
16  from PyQt5.QtGui import (QDoubleValidator, QIntValidator, QKeyEvent,
17                           QKeySequence)
18  from PyQt5.QtWidgets import (QCheckBox, QGroupBox, QHBoxLayout, QLabel,
19                               QLayout, QLineEdit, QRadioButton, QShortcut,
20                               QSizePolicy, QSpacerItem, QVBoxLayout)
21
22  from lintrans.global_settings import (GlobalSettings, GlobalSettingsData,
23                                        UpdateType)
24  from lintrans.gui.dialogs.misc import FixedSizeDialog
25  from lintrans.gui.settings import DisplaySettings
26
27
28  class SettingsDialog(FixedSizeDialog):
29      """An abstract superclass for other simple dialogs."""
30
31      def __init__(self, *args, resettable: bool, **kwargs):
32          """Create the widgets and layout of the dialog, passing ``*args`` and ``**kwargs`` to super."""
33          super().__init__(*args, **kwargs)
34
35          # === Create the widgets
36
37          self._button_confirm = QtWidgets.QPushButton(self)
38          self._button_confirm.setText('Confirm')
39          self._button_confirm.clicked.connect(self._confirm_settings)
40          self._button_confirm.setToolTip('Confirm these new settings<br><b>(Ctrl + Enter)</b>')
41          QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self._button_confirm.click)
42
43          self._button_cancel = QtWidgets.QPushButton(self)
44          self._button_cancel.setText('Cancel')
45          self._button_cancel.clicked.connect(self.reject)
46          self._button_cancel.setToolTip('Revert these settings<br><b>(Escape)</b>')
47
48          if resettable:
49              self._button_reset = QtWidgets.QPushButton(self)
50              self._button_reset.setText('Reset to defaults')
51              self._button_reset.clicked.connect(self._reset_settings)
52              self._button_reset.setToolTip('Reset these settings to their defaults<br><b>(Ctrl + R)</b>')
53              QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(self._button_reset.click)
54
55          # === Arrange the widgets
56
57          self.setContentsMargins(10, 10, 10, 10)
58
59          self._hlay_buttons = QHBoxLayout()
```

```
60              self._hlay_buttons.setSpacing(20)
61
62              if resettable:
63                  self._hlay_buttons.addWidget(self._button_reset)
64
65              self._hlay_buttons.addItem(QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum))
66              self._hlay_buttons.addWidget(self._button_cancel)
67              self._hlay_buttons.addWidget(self._button_confirm)
68
69          def _setup_layout(self, options_layout: QLayout) -> None:
70              """Set the layout of the settings widget.
71
72              .. note:: This method must be called at the end of :meth:`__init__`
73                 in subclasses to setup the layout properly.
74              """
75              vlay_all = QVBoxLayout()
76              vlay_all.setSpacing(20)
77              vlay_all.addLayout(options_layout)
78              vlay_all.addLayout(self._hlay_buttons)
79
80              self.setLayout(vlay_all)
81
82          @abc.abstractmethod
83          def _load_settings(self) -> None:
84              """Load the current settings into the widgets."""
85
86          @abc.abstractmethod
87          def _confirm_settings(self) -> None:
88              """Confirm the settings chosen in the dialog."""
89
90          def _reset_settings(self) -> None:
91              """Reset the settings.
92
93              .. note:: This method is empty but not abstract because not all subclasses will need to implement it.
94              """
95
96
97      class DisplaySettingsDialog(SettingsDialog):
98          """The dialog to allow the user to edit the display settings."""
99
100         def __init__(self, *args, display_settings: DisplaySettings, **kwargs):
101             """Create the widgets and layout of the dialog.
102
103             :param DisplaySettings display_settings: The :class:`~lintrans.gui.settings.DisplaySettings` object to
    ↪  mutate
104             """
105             super().__init__(*args, resettable=True, **kwargs)
106
107             self.display_settings = display_settings
108             self.setWindowTitle('Change display settings')
109
110             self._dict_checkboxes: Dict[str, QCheckBox] = {}
111
112             # === Create the widgets
113
114             # Basic stuff
115
116             self._checkbox_draw_background_grid = QCheckBox(self)
117             self._checkbox_draw_background_grid.setText('Draw &background grid')
118             self._checkbox_draw_background_grid.setToolTip(
119                 'Draw the background grid (axes are always drawn)'
120             )
121             self._dict_checkboxes['b'] = self._checkbox_draw_background_grid
122
123             self._checkbox_draw_transformed_grid = QCheckBox(self)
124             self._checkbox_draw_transformed_grid.setText('Draw t&ransformed grid')
125             self._checkbox_draw_transformed_grid.setToolTip(
126                 'Draw the transformed grid (vectors are handled separately)'
127             )
128             self._dict_checkboxes['r'] = self._checkbox_draw_transformed_grid
129
130             self._checkbox_draw_basis_vectors = QCheckBox(self)
131             self._checkbox_draw_basis_vectors.setText('Draw basis &vectors')
```

```
132             self._checkbox_draw_basis_vectors.setToolTip(
133                 'Draw the transformed basis vectors'
134             )
135             self._checkbox_draw_basis_vectors.clicked.connect(self._update_gui)
136             self._dict_checkboxes['v'] = self._checkbox_draw_basis_vectors
137 
138             self._checkbox_label_basis_vectors = QCheckBox(self)
139             self._checkbox_label_basis_vectors.setText('Label the bas&is vectors')
140             self._checkbox_label_basis_vectors.setToolTip(
141                 'Label the transformed i and j basis vectors'
142             )
143             self._dict_checkboxes['i'] = self._checkbox_label_basis_vectors
144 
145             # Animations
146 
147             self._checkbox_smoothen_determinant = QCheckBox(self)
148             self._checkbox_smoothen_determinant.setText('&Smoothen determinant')
149             self._checkbox_smoothen_determinant.setToolTip(
150                 'Smoothly animate the determinant transition during animation (if possible)'
151             )
152             self._dict_checkboxes['s'] = self._checkbox_smoothen_determinant
153 
154             self._checkbox_applicative_animation = QCheckBox(self)
155             self._checkbox_applicative_animation.setText('&Applicative animation')
156             self._checkbox_applicative_animation.setToolTip(
157                 'Animate the new transformation applied to the current one,\n'
158                 'rather than just that transformation on its own'
159             )
160             self._dict_checkboxes['a'] = self._checkbox_applicative_animation
161 
162             label_animation_time = QLabel(self)
163             label_animation_time.setText('Total animation length (ms)')
164             label_animation_time.setToolTip(
165                 'How long it takes for an animation to complete'
166             )
167 
168             self._lineedit_animation_time = QLineEdit(self)
169             self._lineedit_animation_time.setValidator(QIntValidator(1, 9999, self))
170             self._lineedit_animation_time.textChanged.connect(self._update_gui)
171 
172             label_animation_pause_length = QLabel(self)
173             label_animation_pause_length.setText('Animation pause length (ms)')
174             label_animation_pause_length.setToolTip(
175                 'How many milliseconds to pause for in comma-separated animations'
176             )
177 
178             self._lineedit_animation_pause_length = QLineEdit(self)
179             self._lineedit_animation_pause_length.setValidator(QIntValidator(1, 999, self))
180 
181             # Matrix info
182 
183             self._checkbox_draw_determinant_parallelogram = QCheckBox(self)
184             self._checkbox_draw_determinant_parallelogram.setText('Draw &determinant parallelogram')
185             self._checkbox_draw_determinant_parallelogram.setToolTip(
186                 'Shade the parallelogram representing the determinant of the matrix'
187             )
188             self._checkbox_draw_determinant_parallelogram.clicked.connect(self._update_gui)
189             self._dict_checkboxes['d'] = self._checkbox_draw_determinant_parallelogram
190 
191             self._checkbox_show_determinant_value = QCheckBox(self)
192             self._checkbox_show_determinant_value.setText('Show de&terminant value')
193             self._checkbox_show_determinant_value.setToolTip(
194                 'Show the value of the determinant inside the parallelogram'
195             )
196             self._dict_checkboxes['t'] = self._checkbox_show_determinant_value
197 
198             self._checkbox_draw_eigenvectors = QCheckBox(self)
199             self._checkbox_draw_eigenvectors.setText('Draw &eigenvectors')
200             self._checkbox_draw_eigenvectors.setToolTip('Draw the eigenvectors of the transformations')
201             self._dict_checkboxes['e'] = self._checkbox_draw_eigenvectors
202 
203             self._checkbox_draw_eigenlines = QCheckBox(self)
204             self._checkbox_draw_eigenlines.setText('Draw eigen&lines')
```

```
205             self._checkbox_draw_eigenlines.setToolTip('Draw the eigenlines (invariant lines) of the transformations')
206             self._dict_checkboxes['l'] = self._checkbox_draw_eigenlines
207
208             # Polygon
209
210             self._checkbox_draw_untransformed_polygon = QCheckBox(self)
211             self._checkbox_draw_untransformed_polygon.setText('&Untransformed polygon')
212             self._checkbox_draw_untransformed_polygon.setToolTip('Draw the untransformed version of the polygon')
213             self._dict_checkboxes['u'] = self._checkbox_draw_untransformed_polygon
214
215             self._checkbox_draw_transformed_polygon = QCheckBox(self)
216             self._checkbox_draw_transformed_polygon.setText('Transformed &polygon')
217             self._checkbox_draw_transformed_polygon.setToolTip('Draw the transformed version of the polygon')
218             self._dict_checkboxes['p'] = self._checkbox_draw_transformed_polygon
219
220             # Input/output vectors
221
222             self._checkbox_draw_input_vector = QCheckBox(self)
223             self._checkbox_draw_input_vector.setText('Draw the i&nput vector')
224             self._checkbox_draw_input_vector.setToolTip('Draw the input vector (only in the viewport)')
225             self._dict_checkboxes['n'] = self._checkbox_draw_input_vector
226
227             self._checkbox_draw_output_vector = QCheckBox(self)
228             self._checkbox_draw_output_vector.setText('Draw the &output vector')
229             self._checkbox_draw_output_vector.setToolTip('Draw the output vector (only in the viewport)')
230             self._dict_checkboxes['o'] = self._checkbox_draw_output_vector
231
232             # === Arrange the widgets in QGroupBoxes
233
234             # Basic stuff
235
236             vlay_groupbox_basic_stuff = QVBoxLayout()
237             vlay_groupbox_basic_stuff.setSpacing(20)
238             vlay_groupbox_basic_stuff.addWidget(self._checkbox_draw_background_grid)
239             vlay_groupbox_basic_stuff.addWidget(self._checkbox_draw_transformed_grid)
240             vlay_groupbox_basic_stuff.addWidget(self._checkbox_draw_basis_vectors)
241             vlay_groupbox_basic_stuff.addWidget(self._checkbox_label_basis_vectors)
242
243             groupbox_basic_stuff = QGroupBox('Basic stuff', self)
244             groupbox_basic_stuff.setLayout(vlay_groupbox_basic_stuff)
245
246             # Animations
247
248             hlay_animation_time = QHBoxLayout()
249             hlay_animation_time.addWidget(label_animation_time)
250             hlay_animation_time.addWidget(self._lineedit_animation_time)
251
252             hlay_animation_pause_length = QHBoxLayout()
253             hlay_animation_pause_length.addWidget(label_animation_pause_length)
254             hlay_animation_pause_length.addWidget(self._lineedit_animation_pause_length)
255
256             vlay_groupbox_animations = QVBoxLayout()
257             vlay_groupbox_animations.setSpacing(20)
258             vlay_groupbox_animations.addWidget(self._checkbox_smoothen_determinant)
259             vlay_groupbox_animations.addWidget(self._checkbox_applicative_animation)
260             vlay_groupbox_animations.addLayout(hlay_animation_time)
261             vlay_groupbox_animations.addLayout(hlay_animation_pause_length)
262
263             groupbox_animations = QGroupBox('Animations', self)
264             groupbox_animations.setLayout(vlay_groupbox_animations)
265
266             # Matrix info
267
268             vlay_groupbox_matrix_info = QVBoxLayout()
269             vlay_groupbox_matrix_info.setSpacing(20)
270             vlay_groupbox_matrix_info.addWidget(self._checkbox_draw_determinant_parallelogram)
271             vlay_groupbox_matrix_info.addWidget(self._checkbox_show_determinant_value)
272             vlay_groupbox_matrix_info.addWidget(self._checkbox_draw_eigenvectors)
273             vlay_groupbox_matrix_info.addWidget(self._checkbox_draw_eigenlines)
274
275             groupbox_matrix_info = QGroupBox('Matrix info', self)
276             groupbox_matrix_info.setLayout(vlay_groupbox_matrix_info)
277
```

```
278             # Polygon
279
280             vlay_groupbox_polygon = QVBoxLayout()
281             vlay_groupbox_polygon.setSpacing(20)
282             vlay_groupbox_polygon.addWidget(self._checkbox_draw_untransformed_polygon)
283             vlay_groupbox_polygon.addWidget(self._checkbox_draw_transformed_polygon)
284
285             groupbox_polygon = QGroupBox('Polygon', self)
286             groupbox_polygon.setLayout(vlay_groupbox_polygon)
287
288             # Input/output vectors
289
290             vlay_groupbox_io_vectors = QVBoxLayout()
291             vlay_groupbox_io_vectors.setSpacing(20)
292             vlay_groupbox_io_vectors.addWidget(self._checkbox_draw_input_vector)
293             vlay_groupbox_io_vectors.addWidget(self._checkbox_draw_output_vector)
294
295             groupbox_io_vectors = QGroupBox('Input/output vectors', self)
296             groupbox_io_vectors.setLayout(vlay_groupbox_io_vectors)
297
298             # Now arrange the groupboxes
299             vlay_left = QVBoxLayout()
300             vlay_left.setSpacing(20)
301             vlay_left.addWidget(groupbox_basic_stuff)
302             vlay_left.addWidget(groupbox_animations)
303
304             vlay_right = QVBoxLayout()
305             vlay_right.setSpacing(20)
306             vlay_right.addWidget(groupbox_matrix_info)
307             vlay_right.addWidget(groupbox_polygon)
308             vlay_right.addWidget(groupbox_io_vectors)
309
310             options_layout = QHBoxLayout()
311             options_layout.setSpacing(20)
312             options_layout.addLayout(vlay_left)
313             options_layout.addLayout(vlay_right)
314
315             self._setup_layout(options_layout)
316
317             # Finally, we load the current settings and update the GUI
318             self._load_settings()
319             self._update_gui()
320
321         def _load_settings(self) -> None:
322             """Load the current display settings into the widgets."""
323             # Basic stuff
324             self._checkbox_draw_background_grid.setChecked(self.display_settings.draw_background_grid)
325             self._checkbox_draw_transformed_grid.setChecked(self.display_settings.draw_transformed_grid)
326             self._checkbox_draw_basis_vectors.setChecked(self.display_settings.draw_basis_vectors)
327             self._checkbox_label_basis_vectors.setChecked(self.display_settings.label_basis_vectors)
328
329             # Animations
330             self._checkbox_smoothen_determinant.setChecked(self.display_settings.smoothen_determinant)
331             self._checkbox_applicative_animation.setChecked(self.display_settings.applicative_animation)
332             self._lineedit_animation_time.setText(str(self.display_settings.animation_time))
333             self._lineedit_animation_pause_length.setText(str(self.display_settings.animation_pause_length))
334
335             # Matrix info
336             self._checkbox_draw_determinant_parallelogram.setChecked(
                ↪  self.display_settings.draw_determinant_parallelogram)
337             self._checkbox_show_determinant_value.setChecked(self.display_settings.show_determinant_value)
338             self._checkbox_draw_eigenvectors.setChecked(self.display_settings.draw_eigenvectors)
339             self._checkbox_draw_eigenlines.setChecked(self.display_settings.draw_eigenlines)
340
341             # Polygon
342             self._checkbox_draw_untransformed_polygon.setChecked(self.display_settings.draw_untransformed_polygon)
343             self._checkbox_draw_transformed_polygon.setChecked(self.display_settings.draw_transformed_polygon)
344
345             # Input/output vectors
346             self._checkbox_draw_input_vector.setChecked(self.display_settings.draw_input_vector)
347             self._checkbox_draw_output_vector.setChecked(self.display_settings.draw_output_vector)
348
349         def _confirm_settings(self) -> None:
```

```
350             """Build a :class:`~lintrans.gui.settings.DisplaySettings` object and assign it."""
351             # Basic stuff
352             self.display_settings.draw_background_grid = self._checkbox_draw_background_grid.isChecked()
353             self.display_settings.draw_transformed_grid = self._checkbox_draw_transformed_grid.isChecked()
354             self.display_settings.draw_basis_vectors = self._checkbox_draw_basis_vectors.isChecked()
355             self.display_settings.label_basis_vectors = self._checkbox_label_basis_vectors.isChecked()
356
357             # Animations
358             self.display_settings.smoothen_determinant = self._checkbox_smoothen_determinant.isChecked()
359             self.display_settings.applicative_animation = self._checkbox_applicative_animation.isChecked()
360             self.display_settings.animation_time = int(self._lineedit_animation_time.text())
361             self.display_settings.animation_pause_length = int(self._lineedit_animation_pause_length.text())
362
363             # Matrix info
364             self.display_settings.draw_determinant_parallelogram =
                ↪  self._checkbox_draw_determinant_parallelogram.isChecked()
365             self.display_settings.show_determinant_value = self._checkbox_show_determinant_value.isChecked()
366             self.display_settings.draw_eigenvectors = self._checkbox_draw_eigenvectors.isChecked()
367             self.display_settings.draw_eigenlines = self._checkbox_draw_eigenlines.isChecked()
368
369             # Polygon
370             self.display_settings.draw_untransformed_polygon = self._checkbox_draw_untransformed_polygon.isChecked()
371             self.display_settings.draw_transformed_polygon = self._checkbox_draw_transformed_polygon.isChecked()
372
373             # Input/output vectors
374             self.display_settings.draw_input_vector = self._checkbox_draw_input_vector.isChecked()
375             self.display_settings.draw_output_vector = self._checkbox_draw_output_vector.isChecked()
376
377             self.accept()
378
379         def _reset_settings(self) -> None:
380             """Reset the display settings to their defaults."""
381             self.display_settings = DisplaySettings()
382             self._load_settings()
383             self._update_gui()
384
385         def _update_gui(self) -> None:
386             """Update the GUI according to other widgets in the GUI.
387
388             For example, this method updates which checkboxes are enabled based on the values of other checkboxes.
389             """
390             self._checkbox_show_determinant_value.setEnabled(self._checkbox_draw_determinant_parallelogram.isChecked())
391             self._checkbox_label_basis_vectors.setEnabled(self._checkbox_draw_basis_vectors.isChecked())
392
393             try:
394                 self._button_confirm.setEnabled(int(self._lineedit_animation_time.text()) >= 10)
395             except ValueError:
396                 self._button_confirm.setEnabled(False)
397
398         def keyPressEvent(self, event: QKeyEvent) -> None:
399             """Handle a :class:`QKeyEvent` by manually activating toggling checkboxes.
400
401             Qt handles these shortcuts automatically and allows the user to do ``Alt + Key``
402             to activate a simple shortcut defined with ``&``. However, I like to be able to
403             just hit ``Key`` and have the shortcut activate.
404             """
405             letter = event.text().lower()
406             key = event.key()
407
408             if letter in self._dict_checkboxes:
409                 self._dict_checkboxes[letter].animateClick()
410
411             # Return or keypad enter
412             elif key == Qt.Key_Return or key == Qt.Key_Enter:
413                 self._button_confirm.click()
414
415             # Escape
416             elif key == Qt.Key_Escape:
417                 self._button_cancel.click()
418
419             else:
420                 event.ignore()
421                 return
```

```
422
423             event.accept()
424
425
426     class GlobalSettingsDialog(SettingsDialog):
427         """The dialog to allow the user to edit the display settings."""
428
429         def __init__(self, *args, **kwargs):
430             """Create the widgets and layout of the dialog."""
431             super().__init__(*args, resettable=True, **kwargs)
432
433             self._data: GlobalSettingsData = GlobalSettings().get_data()
434             self.setWindowTitle('Change global settings')
435
436             # === Create the widgets
437
438             groupbox_update_types = QGroupBox('Update prompt type', self)
439             self._radio_button_auto = QRadioButton('Always update automatically', groupbox_update_types)
440             self._radio_button_prompt = QRadioButton('Always ask to update', groupbox_update_types)
441             self._radio_button_never = QRadioButton('Never update', groupbox_update_types)
442
443             label_cursor_epsilon = QLabel(self)
444             label_cursor_epsilon.setText('Cursor drag proximity (pixels)')
445             label_cursor_epsilon.setToolTip(
446                 'The maximum distance (in pixels) from a draggable point before it will be dragged'
447             )
448
449             self._lineedit_cursor_epsilon = QLineEdit(self)
450             self._lineedit_cursor_epsilon.setValidator(QIntValidator(1, 99, self))
451             self._lineedit_cursor_epsilon.setText(str(self._data.cursor_epsilon))
452             self._lineedit_cursor_epsilon.textChanged.connect(self._update_gui)
453
454             self._checkbox_snap_to_int_coords = QCheckBox(self)
455             self._checkbox_snap_to_int_coords.setText('Snap to integer coordinates')
456             self._checkbox_snap_to_int_coords.setToolTip(
457                 'Whether vectors should snap the integer coordinates when dragging them'
458             )
459             self._checkbox_snap_to_int_coords.clicked.connect(self._update_gui)
460
461             label_snap_dist = QLabel(self)
462             label_snap_dist.setText('Snap distance (grid units)')
463             label_snap_dist.setToolTip(
464                 'The minimum distacne (in grid units) that a draggable point '
465                 'must be from an integer coordinate to snap to it'
466             )
467
468             self._lineedit_snap_dist = QLineEdit(self)
469             self._lineedit_snap_dist.setValidator(QDoubleValidator(0.0, 0.99, 2, self))
470             self._lineedit_snap_dist.setText(str(self._data.snap_dist))
471             self._lineedit_snap_dist.textChanged.connect(self._update_gui)
472
473             # === Arrange the widgets
474
475             vlay_update_type = QVBoxLayout()
476             vlay_update_type.addWidget(self._radio_button_auto)
477             vlay_update_type.addWidget(self._radio_button_prompt)
478             vlay_update_type.addWidget(self._radio_button_never)
479             groupbox_update_types.setLayout(vlay_update_type)
480
481             hlay_cursor_epsilon = QHBoxLayout()
482             hlay_cursor_epsilon.addWidget(label_cursor_epsilon)
483             hlay_cursor_epsilon.addWidget(self._lineedit_cursor_epsilon)
484
485             hlay_snap_dist = QHBoxLayout()
486             hlay_snap_dist.addWidget(label_snap_dist)
487             hlay_snap_dist.addWidget(self._lineedit_snap_dist)
488
489             vlay_dist = QVBoxLayout()
490             vlay_dist.setSpacing(20)
491             vlay_dist.addLayout(hlay_cursor_epsilon)
492             vlay_dist.addWidget(self._checkbox_snap_to_int_coords)
493             vlay_dist.addLayout(hlay_snap_dist)
494
```

```
495             groupbox_dist = QGroupBox('Distances', self)
496             groupbox_dist.setLayout(vlay_dist)
497
498             options_layout = QVBoxLayout()
499             options_layout.setSpacing(20)
500             options_layout.addWidget(groupbox_update_types)
501             options_layout.addWidget(groupbox_dist)
502
503             self._load_settings()
504             self._update_gui()
505             self._setup_layout(options_layout)
506
507         def _update_gui(self) -> None:
508             """Update the GUI according to other widgets in the GUI."""
509             if self._lineedit_cursor_epsilon.text() == '':
510                 cursor_epsilon = False
511             else:
512                 cursor_epsilon = 0 <= int(self._lineedit_cursor_epsilon.text()) <= 99
513
514             if self._lineedit_snap_dist.text() == '':
515                 snap_dist = False
516             else:
517                 snap_dist = 0.0 <= float(self._lineedit_snap_dist.text()) <= 1.0
518
519             self._lineedit_snap_dist.setEnabled(self._checkbox_snap_to_int_coords.isChecked())
520             self._button_confirm.setEnabled(cursor_epsilon and snap_dist)
521
522         def _load_settings(self) -> None:
523             """Load the current display settings into the widgets."""
524             if self._data.update_type == UpdateType.auto:
525                 self._radio_button_auto.setChecked(True)
526             elif self._data.update_type == UpdateType.prompt:
527                 self._radio_button_prompt.setChecked(True)
528             elif self._data.update_type == UpdateType.never:
529                 self._radio_button_never.setChecked(True)
530
531             self._lineedit_cursor_epsilon.setText(str(self._data.cursor_epsilon))
532             self._checkbox_snap_to_int_coords.setChecked(self._data.snap_to_int_coords)
533             self._lineedit_snap_dist.setText(str(self._data.snap_dist))
534
535         def _confirm_settings(self) -> None:
536             """Set the global settings."""
537             if self._radio_button_auto.isChecked():
538                 self._data.update_type = UpdateType.auto
539             elif self._radio_button_prompt.isChecked():
540                 self._data.update_type = UpdateType.prompt
541             elif self._radio_button_never.isChecked():
542                 self._data.update_type = UpdateType.never
543
544             self._data.cursor_epsilon = int(self._lineedit_cursor_epsilon.text())
545             self._data.snap_to_int_coords = self._checkbox_snap_to_int_coords.isChecked()
546             self._data.snap_dist = float(self._lineedit_snap_dist.text())
547
548             GlobalSettings().set_data(self._data)
549
550             self.accept()
551
552         def _reset_settings(self) -> None:
553             """Reset the internal data values to their defaults."""
554             self._data = GlobalSettingsData()
555             self._load_settings()
556             self._update_gui()
```

A.19 gui/dialogs/__init__.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
```

```

    """This package provides separate dialogs for the main GUI.

    These dialogs are for defining new matrices in different ways and editing settings.
    """

    from .define_new_matrix import (DefineAsExpressionDialog, DefineMatrixDialog,
                                    DefineNumericallyDialog, DefineVisuallyDialog)
    from .misc import (AboutDialog, DefinePolygonDialog, FileSelectDialog,
                       InfoPanelDialog, PromptUpdateDialog)
    from .settings import DisplaySettingsDialog

    __all__ = ['AboutDialog', 'DefineAsExpressionDialog', 'DefineMatrixDialog',
               'DefineNumericallyDialog', 'DefinePolygonDialog', 'DefineVisuallyDialog',
               'DisplaySettingsDialog', 'FileSelectDialog', 'InfoPanelDialog', 'PromptUpdateDialog']
```

A.20 gui/dialogs/define_new_matrix.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 an abstract :class:`DefineMatrixDialog` class and subclasses."""

    from __future__ import annotations

    import abc
    from typing import List, Tuple

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

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

    _ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'


    def get_first_undefined_matrix(wrapper: MatrixWrapper) -> str:
        """Return the letter of the first undefined matrix in the given wrapper, or ``A`` if all matrices are
    ↪   defined."""
        defined_matrices = [x for x, _ in wrapper.get_defined_matrices()]
        for letter in _ALPHABET_NO_I:
            if letter not in defined_matrices:
                return letter

        return 'A'


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

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

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

```

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

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

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

            # === Create the widgets

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

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

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

            self._combobox_letter = QtWidgets.QComboBox(self)

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

            self._combobox_letter.activated.connect(self._load_matrix)
            self._combobox_letter.setCurrentText(get_first_undefined_matrix(self.matrix_wrapper))

            # === Arrange the widgets

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

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

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

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

            self.setLayout(self._vlay_all)

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

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

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

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

```
125
126             We have this method in the superclass so that we can define it as the slot
127             for the :meth:`QComboBox.activated` signal in this constructor, rather than
128             having to define that in the constructor of every subclass.
129             """
130
131         @abc.abstractmethod
132         @pyqtSlot()
133         def _confirm_matrix(self) -> None:
134             """Confirm the inputted matrix and assign it.
135
136             .. note:: When subclassing, this method should mutate ``self.matrix_wrapper`` and then call
    ↪  ``self.accept()``.
137             """
138
139
140     class DefineVisuallyDialog(DefineMatrixDialog):
141         """The dialog class that allows the user to define a matrix visually."""
142
143         def __init__(
144                 self,
145                 *args,
146                 matrix_wrapper: MatrixWrapper,
147                 display_settings: DisplaySettings,
148                 polygon_points: List[Tuple[float, float]],
149                 input_vector: Tuple[float, float],
150                 **kwargs
151         ):
152             """Create the widgets and layout of the dialog.
153
154             :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
155             """
156             super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
157
158             self.setMinimumSize(700, 550)
159
160             # === Create the widgets
161
162             self._plot = DefineMatrixVisuallyWidget(
163                 self,
164                 display_settings=display_settings,
165                 polygon_points=polygon_points,
166                 input_vector=input_vector
167             )
168
169             # === Arrange the widgets
170
171             self._hlay_definition.addWidget(self._plot)
172             self._hlay_definition.setStretchFactor(self._plot, 1)
173
174             self._vlay_all.addLayout(self._hlay_definition)
175             self._vlay_all.addLayout(self._hlay_buttons)
176
177             # We load the default matrix A into the plot
178             self._load_matrix(0)
179
180             # We also enable the confirm button, because any visually defined matrix is valid
181             self._button_confirm.setEnabled(True)
182
183         @pyqtSlot()
184         def _update_confirm_button(self) -> None:
185             """Enable the confirm button.
186
187             .. note::
188                The confirm button is always enabled in this dialog and this method is never actually used,
189                so it's got an empty body. It's only here because we need to implement the abstract method.
190             """
191
192         @pyqtSlot(int)
193         def _load_matrix(self, index: int) -> None:
194             """Show the selected matrix on the plot. If the matrix is None, show the identity."""
195             matrix = self.matrix_wrapper[self._selected_letter]
196
```

```
197             if matrix is None:
198                 self._plot.plot_matrix(eye(2))
199             else:
200                 self._plot.plot_matrix(matrix)
201 
202             self._plot.update()
203 
204         @pyqtSlot()
205         def _confirm_matrix(self) -> None:
206             """Confirm the matrix that's been defined visually."""
207             matrix: MatrixType = array([
208                 [self._plot.point_i[0], self._plot.point_j[0]],
209                 [self._plot.point_i[1], self._plot.point_j[1]]
210             ])
211 
212             self.matrix_wrapper[self._selected_letter] = matrix
213             self.accept()
214 
215 
216     class DefineNumericallyDialog(DefineMatrixDialog):
217         """The dialog class that allows the user to define a new matrix numerically."""
218 
219         def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
220             """Create the widgets and layout of the dialog.
221 
222             :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
223             """
224             super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
225 
226             # === Create the widgets
227 
228             # tl = top left, br = bottom right, etc.
229             self._element_tl = QLineEdit(self)
230             self._element_tl.textChanged.connect(self._update_confirm_button)
231             self._element_tl.setValidator(QDoubleValidator())
232 
233             self._element_tr = QLineEdit(self)
234             self._element_tr.textChanged.connect(self._update_confirm_button)
235             self._element_tr.setValidator(QDoubleValidator())
236 
237             self._element_bl = QLineEdit(self)
238             self._element_bl.textChanged.connect(self._update_confirm_button)
239             self._element_bl.setValidator(QDoubleValidator())
240 
241             self._element_br = QLineEdit(self)
242             self._element_br.textChanged.connect(self._update_confirm_button)
243             self._element_br.setValidator(QDoubleValidator())
244 
245             self._matrix_elements = (self._element_tl, self._element_tr, self._element_bl, self._element_br)
246 
247             font_parens = self.font()
248             font_parens.setPointSize(int(font_parens.pointSize() * 5))
249             font_parens.setWeight(int(font_parens.weight() / 5))
250 
251             label_paren_left = QLabel(self)
252             label_paren_left.setText('(')
253             label_paren_left.setFont(font_parens)
254 
255             label_paren_right = QLabel(self)
256             label_paren_right.setText(')')
257             label_paren_right.setFont(font_parens)
258 
259             # === Arrange the widgets
260 
261             grid_matrix = QGridLayout()
262             grid_matrix.setSpacing(20)
263             grid_matrix.addWidget(label_paren_left, 0, 0, -1, 1)
264             grid_matrix.addWidget(self._element_tl, 0, 1)
265             grid_matrix.addWidget(self._element_tr, 0, 2)
266             grid_matrix.addWidget(self._element_bl, 1, 1)
267             grid_matrix.addWidget(self._element_br, 1, 2)
268             grid_matrix.addWidget(label_paren_right, 0, 3, -1, 1)
269 
```

```
270             self._hlay_definition.addLayout(grid_matrix)
271
272             self._vlay_all.addLayout(self._hlay_definition)
273             self._vlay_all.addLayout(self._hlay_buttons)
274
275             # We load the default matrix A into the boxes
276             self._load_matrix(0)
277
278             self._element_tl.setFocus()
279
280         @pyqtSlot()
281         def _update_confirm_button(self) -> None:
282             """Enable the confirm button if there are valid floats in every box."""
283             for elem in self._matrix_elements:
284                 if not is_valid_float(elem.text()):
285                     # If they're not all numbers, then we can't confirm it
286                     self._button_confirm.setEnabled(False)
287                     return
288
289             # If we didn't find anything invalid
290             self._button_confirm.setEnabled(True)
291
292         @pyqtSlot(int)
293         def _load_matrix(self, index: int) -> None:
294             """If the selected matrix is defined, load its values into the boxes."""
295             matrix = self.matrix_wrapper[self._selected_letter]
296
297             if matrix is None:
298                 for elem in self._matrix_elements:
299                     elem.setText('')
300
301             else:
302                 self._element_tl.setText(round_float(matrix[0][0]))
303                 self._element_tr.setText(round_float(matrix[0][1]))
304                 self._element_bl.setText(round_float(matrix[1][0]))
305                 self._element_br.setText(round_float(matrix[1][1]))
306
307             self._update_confirm_button()
308
309         @pyqtSlot()
310         def _confirm_matrix(self) -> None:
311             """Confirm the matrix in the boxes and assign it to the name in the combo box."""
312             matrix: MatrixType = array([
313                 [float(self._element_tl.text()), float(self._element_tr.text())],
314                 [float(self._element_bl.text()), float(self._element_br.text())]
315             ])
316
317             self.matrix_wrapper[self._selected_letter] = matrix
318             self.accept()
319
320
321     class DefineAsExpressionDialog(DefineMatrixDialog):
322         """The dialog class that allows the user to define a matrix as an expression of other matrices."""
323
324         def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
325             """Create the widgets and layout of the dialog.
326
327             :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
328             """
329             super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
330
331             self.setMinimumWidth(450)
332
333             # === Create the widgets
334
335             self._lineedit_expression_box = QLineEdit(self)
336             self._lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
337             self._lineedit_expression_box.textChanged.connect(self._update_confirm_button)
338             self._lineedit_expression_box.setValidator(MatrixExpressionValidator())
339
340             # === Arrange the widgets
341
342             self._hlay_definition.addWidget(self._lineedit_expression_box)
```

```
343
344             self._vlay_all.addLayout(self._hlay_definition)
345             self._vlay_all.addLayout(self._hlay_buttons)
346
347             # Load the matrix if it's defined as an expression
348             self._load_matrix(0)
349
350             self._lineedit_expression_box.setFocus()
351
352         @pyqtSlot()
353         def _update_confirm_button(self) -> None:
354             """Enable the confirm button if the matrix expression is valid in the wrapper."""
355             text = self._lineedit_expression_box.text()
356             valid_expression = self.matrix_wrapper.is_valid_expression(text)
357
358             self._button_confirm.setEnabled(
359                 valid_expression
360                 and self._selected_letter not in text
361                 and self._selected_letter not in self.matrix_wrapper.get_expression_dependencies(text)
362             )
363
364         @pyqtSlot(int)
365         def _load_matrix(self, index: int) -> None:
366             """If the selected matrix is defined an expression, load that expression into the box."""
367             if (expr := self.matrix_wrapper.get_expression(self._selected_letter)) is not None:
368                 self._lineedit_expression_box.setText(expr)
369             else:
370                 self._lineedit_expression_box.setText('')
371
372         @pyqtSlot()
373         def _confirm_matrix(self) -> None:
374             """Evaluate the matrix expression and assign its value to the name in the combo box."""
375             self.matrix_wrapper[self._selected_letter] = self._lineedit_expression_box.text()
376             self.accept()
```

A.21 gui/plots/widgets.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides the actual widgets that can be used to visualize transformations in the GUI."""
8
9   from __future__ import annotations
10
11  import operator
12  from abc import abstractmethod
13  from copy import copy
14  from math import dist
15  from typing import List, Optional, Tuple
16
17  from PyQt5.QtCore import QPointF, Qt, pyqtSlot
18  from PyQt5.QtGui import (QBrush, QColor, QMouseEvent, QPainter, QPaintEvent,
19                           QPen, QPolygonF)
20
21  from lintrans.global_settings import GlobalSettings
22  from lintrans.gui.settings import DisplaySettings
23  from lintrans.typing_ import MatrixType
24
25  from .classes import InteractivePlot, VisualizeTransformationPlot
26
27
28  class VisualizeTransformationWidget(VisualizeTransformationPlot):
29      """This widget is used in the main window to visualize transformations.
30
31      It handles all the rendering itself, and the only method that the user needs to care about
32      is :meth:`plot_matrix`, which allows you to visualize the given matrix transformation.
33      """
```

```
34
35          _COLOUR_OUTPUT_VECTOR = QColor('#f7c216')
36
37          def __init__(self, *args, display_settings: DisplaySettings, polygon_points: List[Tuple[float, float]],
            ↪  **kwargs):
38              """Create the widget and assign its display settings, passing ``*args`` and ``**kwargs`` to super."""
39              super().__init__(*args, **kwargs)
40
41              self.display_settings = display_settings
42              self.polygon_points = polygon_points
43
44          def plot_matrix(self, matrix: MatrixType) -> None:
45              """Plot the given matrix on the grid by setting the basis vectors.
46
47              .. warning:: This method does not call :meth:`QWidget.update()`. This must be done by the caller.
48
49              :param MatrixType matrix: The matrix to plot
50              """
51              self.point_i = (matrix[0][0], matrix[1][0])
52              self.point_j = (matrix[0][1], matrix[1][1])
53
54          def _draw_scene(self, painter: QPainter) -> None:
55              """Draw the default scene of the transformation.
56
57              This method exists to make it easier to split the main viewport from visual definitions while
58              not using multiple :class:`QPainter` objects from a single :meth:`paintEvent` call in a subclass.
59              """
60              painter.setRenderHint(QPainter.Antialiasing)
61              painter.setBrush(Qt.NoBrush)
62
63              self._draw_background(painter, self.display_settings.draw_background_grid)
64
65              if self.display_settings.draw_eigenlines:
66                  self._draw_eigenlines(painter)
67
68              if self.display_settings.draw_eigenvectors:
69                  self._draw_eigenvectors(painter)
70
71              if self.display_settings.draw_determinant_parallelogram:
72                  self._draw_determinant_parallelogram(painter)
73
74                  if self.display_settings.show_determinant_value:
75                      self._draw_determinant_text(painter)
76
77              if self.display_settings.draw_transformed_grid:
78                  self._draw_transformed_grid(painter)
79
80              if self.display_settings.draw_basis_vectors:
81                  self._draw_basis_vectors(painter)
82
83                  if self.display_settings.label_basis_vectors:
84                      self._draw_basis_vector_labels(painter)
85
86              if self.display_settings.draw_untransformed_polygon:
87                  self._draw_untransformed_polygon(painter)
88
89              if self.display_settings.draw_transformed_polygon:
90                  self._draw_transformed_polygon(painter)
91
92          @abstractmethod
93          def paintEvent(self, event: QPaintEvent) -> None:
94              """Paint the scene of the transformation."""
95
96
97      class MainViewportWidget(VisualizeTransformationWidget, InteractivePlot):
98          """This is the widget for the main viewport.
99
100         It extends :class:`VisualizeTransformationWidget` with input and output vectors.
101         """
102
103         def __init__(self, *args, **kwargs):
104             """Create the main viewport widget with its input point."""
105             super().__init__(*args,  **kwargs)
```

```
106
107             self.point_input_vector: Tuple[float, float] = (1, 1)
108             self._dragging_vector: bool = False
109
110         def _draw_input_vector(self, painter: QPainter) -> None:
111             """Draw the input vector."""
112             pen = QPen(QColor('#000000'), self._WIDTH_VECTOR_LINE)
113             painter.setPen(pen)
114
115             x, y = self.canvas_coords(*self.point_input_vector)
116             painter.drawLine(*self._canvas_origin, x, y)
117
118             painter.setBrush(self._BRUSH_SOLID_WHITE)
119             cursor_epsilon = GlobalSettings().get_data().cursor_epsilon
120
121             painter.setPen(Qt.NoPen)
122             painter.drawPie(
123                 x - cursor_epsilon,
124                 y - cursor_epsilon,
125                 2 * cursor_epsilon,
126                 2 * cursor_epsilon,
127                 0,
128                 16 * 360
129             )
130
131             painter.setPen(pen)
132             painter.drawArc(
133                 x - cursor_epsilon,
134                 y - cursor_epsilon,
135                 2 * cursor_epsilon,
136                 2 * cursor_epsilon,
137                 0,
138                 16 * 360
139             )
140
141         def _draw_output_vector(self, painter: QPainter) -> None:
142             """Draw the output vector."""
143             painter.setPen(QPen(self._COLOUR_OUTPUT_VECTOR, self._WIDTH_VECTOR_LINE))
144             painter.setBrush(QBrush(self._COLOUR_OUTPUT_VECTOR, Qt.SolidPattern))
145
146             x, y = self.canvas_coords(*(self._matrix @ self.point_input_vector))
147             cursor_epsilon = GlobalSettings().get_data().cursor_epsilon
148
149             painter.drawLine(*self._canvas_origin, x, y)
150             painter.drawPie(
151                 x - cursor_epsilon,
152                 y - cursor_epsilon,
153                 2 * cursor_epsilon,
154                 2 * cursor_epsilon,
155                 0,
156                 16 * 360
157             )
158
159         def paintEvent(self, event: QPaintEvent) -> None:
160             """Paint the scene by just calling :meth:`_draw_scene` and drawing the I/O vectors."""
161             painter = QPainter()
162             painter.begin(self)
163
164             self._draw_scene(painter)
165
166             if self.display_settings.draw_output_vector:
167                 self._draw_output_vector(painter)
168
169             if self.display_settings.draw_input_vector:
170                 self._draw_input_vector(painter)
171
172             painter.end()
173             event.accept()
174
175         def mousePressEvent(self, event: QMouseEvent) -> None:
176             """Check if the user has clicked on the input vector."""
177             cursor_pos = (event.x(), event.y())
178
```

```
179             if event.button() != Qt.LeftButton:
180                 event.ignore()
181                 return
182 
183             if self._is_within_epsilon(cursor_pos, self.point_input_vector):
184                 self._dragging_vector = True
185 
186             event.accept()
187 
188         def mouseReleaseEvent(self, event: QMouseEvent) -> None:
189             """Stop dragging the input vector."""
190             if event.button() == Qt.LeftButton:
191                 self._dragging_vector = False
192                 event.accept()
193             else:
194                 event.ignore()
195 
196         def mouseMoveEvent(self, event: QMouseEvent) -> None:
197             """Drag the input vector if the user has clicked on it."""
198             if not self._dragging_vector:
199                 event.ignore()
200                 return
201 
202             x, y = self._round_to_int_coord(self._grid_coords(event.x(), event.y()))
203             self.point_input_vector = (x, y)
204 
205             self.update()
206             event.accept()
207 
208 
209     class DefineMatrixVisuallyWidget(VisualizeTransformationWidget, InteractivePlot):
210         """This widget allows the user to visually define a matrix.
211 
212         This is just the widget itself. If you want the dialog, use
213         :class:`~lintrans.gui.dialogs.define_new_matrix.DefineVisuallyDialog`.
214         """
215 
216         def __init__(
217             self,
218             *args,
219             display_settings: DisplaySettings,
220             polygon_points: List[Tuple[float, float]],
221             input_vector: Tuple[float, float],
222             **kwargs
223         ) -> None:
224             """Create the widget and enable mouse tracking. ``*args`` and ``**kwargs`` are passed to ``super()``."""
225             super().__init__(*args, display_settings=display_settings, polygon_points=polygon_points, **kwargs)
226 
227             self._input_vector = input_vector
228             self._dragged_point: Tuple[float, float] | None = None
229 
230         def _draw_input_vector(self, painter: QPainter) -> None:
231             """Draw the input vector."""
232             color = QColor('#000000')
233             color.setAlpha(0x88)
234             pen = QPen(color, self._WIDTH_VECTOR_LINE)
235             painter.setPen(pen)
236 
237             x, y = self.canvas_coords(*self._input_vector)
238             painter.drawLine(*self._canvas_origin, x, y)
239 
240             painter.setBrush(self._BRUSH_SOLID_WHITE)
241             cursor_epsilon = GlobalSettings().get_data().cursor_epsilon
242 
243             painter.setPen(Qt.NoPen)
244             painter.drawPie(
245                 x - cursor_epsilon,
246                 y - cursor_epsilon,
247                 2 * cursor_epsilon,
248                 2 * cursor_epsilon,
249                 0,
250                 16 * 360
251             )
```

```
252
253             painter.setPen(pen)
254             painter.drawArc(
255                 x - cursor_epsilon,
256                 y - cursor_epsilon,
257                 2 * cursor_epsilon,
258                 2 * cursor_epsilon,
259                 0,
260                 16 * 360
261             )
262
263         def _draw_output_vector(self, painter: QPainter) -> None:
264             """Draw the output vector."""
265             color = copy(self._COLOUR_OUTPUT_VECTOR)
266             color.setAlpha(0x88)
267             painter.setPen(QPen(color, self._WIDTH_VECTOR_LINE))
268             painter.setBrush(QBrush(self._COLOUR_OUTPUT_VECTOR, Qt.SolidPattern))
269
270             x, y = self.canvas_coords(*(self._matrix @ self._input_vector))
271             cursor_epsilon = GlobalSettings().get_data().cursor_epsilon
272
273             painter.drawLine(*self._canvas_origin, x, y)
274             painter.drawPie(
275                 x - cursor_epsilon,
276                 y - cursor_epsilon,
277                 2 * cursor_epsilon,
278                 2 * cursor_epsilon,
279                 0,
280                 16 * 360
281             )
282
283         def paintEvent(self, event: QPaintEvent) -> None:
284             """Paint the scene by just calling :meth:`_draw_scene`."""
285             painter = QPainter()
286             painter.begin(self)
287
288             self._draw_scene(painter)
289
290             if self.display_settings.draw_output_vector:
291                 self._draw_output_vector(painter)
292
293             if self.display_settings.draw_input_vector:
294                 self._draw_input_vector(painter)
295
296             painter.end()
297             event.accept()
298
299         def mousePressEvent(self, event: QMouseEvent) -> None:
300             """Set the dragged point if the cursor is within the cursor epsilon.
301
302             See :attr:`lintrans.global_settings.GlobalSettingsData.cursor_epsilon`.
303             """
304             cursor_pos = (event.x(), event.y())
305
306             if event.button() != Qt.LeftButton:
307                 event.ignore()
308                 return
309
310             for point in (self.point_i, self.point_j):
311                 if self._is_within_epsilon(cursor_pos, point):
312                     self._dragged_point = point[0], point[1]
313
314             event.accept()
315
316         def mouseReleaseEvent(self, event: QMouseEvent) -> None:
317             """Handle the mouse click being released by unsetting the dragged point."""
318             if event.button() == Qt.LeftButton:
319                 self._dragged_point = None
320                 event.accept()
321             else:
322                 event.ignore()
323
324         def mouseMoveEvent(self, event: QMouseEvent) -> None:
```

```
325             """Handle the mouse moving on the canvas."""
326             if self._dragged_point is None:
327                 event.ignore()
328                 return
329 
330             x, y = self._round_to_int_coord(self._grid_coords(event.x(), event.y()))
331 
332             if self._dragged_point == self.point_i:
333                 self.point_i = x, y
334 
335             elif self._dragged_point == self.point_j:
336                 self.point_j = x, y
337 
338             self._dragged_point = x, y
339 
340             self.update()
341             event.accept()
342 
343 
344     class DefinePolygonWidget(InteractivePlot):
345         """This widget allows the user to define a polygon by clicking and dragging points on the canvas."""
346 
347         def __init__(self, *args, polygon_points: List[Tuple[float, float]], **kwargs):
348             """Create the widget with a list of points and a dragged point index."""
349             super().__init__(*args, **kwargs)
350 
351             self._dragged_point_index: Optional[int] = None
352             self.points = polygon_points.copy()
353 
354         @pyqtSlot()
355         def reset_polygon(self) -> None:
356             """Reset the polygon and update the widget."""
357             self.points = []
358             self.update()
359 
360         def mousePressEvent(self, event: QMouseEvent) -> None:
361             """Handle the mouse being clicked by adding a point or setting the dragged point index to an existing
    ↪  point."""
362             if event.button() not in (Qt.LeftButton, Qt.RightButton):
363                 event.ignore()
364                 return
365 
366             canvas_pos = (event.x(), event.y())
367             grid_pos = self._grid_coords(*canvas_pos)
368 
369             if event.button() == Qt.LeftButton:
370                 for i, point in enumerate(self.points):
371                     if self._is_within_epsilon(canvas_pos, point):
372                         self._dragged_point_index = i
373                         event.accept()
374                         return
375 
376                 new_point = self._round_to_int_coord(grid_pos)
377 
378                 if len(self.points) < 2:
379                     self.points.append(new_point)
380                     self._dragged_point_index = -1
381                 else:
382                     # FIXME: This algorithm doesn't work very well when the new point is far away
383                     # from the existing polygon; it just picks the longest side
384 
385                     # Get a list of line segments and a list of their lengths
386                     line_segments = list(zip(self.points, self.points[1:])) + [(self.points[-1], self.points[0])]
387                     segment_lengths = map(lambda t: dist(*t), line_segments)
388 
389                     # Get the distance from each point in the polygon to the new point
390                     distances_to_point = [dist(p, new_point) for p in self.points]
391 
392                     # For each pair of list-adjacent points, zip their distances to
393                     # the new point into a tuple, and add them together
394                     # This gives us the lengths of the catheti of the triangles that
395                     # connect the new point to each pair of adjacent points
396                     dist_to_point_pairs = list(zip(distances_to_point, distances_to_point[1:])) + \
```

```
                    [(distances_to_point[-1], distances_to_point[0])]

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

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

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

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

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

        self.update()
        event.accept()

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

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

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

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

        self.update()

        event.accept()

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

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

        painter.setBrush(self._BRUSH_SOLID_WHITE)
        cursor_epsilon = GlobalSettings().get_data().cursor_epsilon

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

            painter.setPen(Qt.NoPen)
            painter.drawPie(
                x - cursor_epsilon,
                y - cursor_epsilon,
                2 * cursor_epsilon,
```

```
                2 * cursor_epsilon,
                0,
                16 * 360
            )

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

        painter.setBrush(Qt.NoBrush)

    def paintEvent(self, event: QPaintEvent) -> None:
        """Draw the polygon on the canvas."""
        painter = QPainter()
        painter.begin(self)

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

        self._draw_background(painter, True)

        self._draw_polygon(painter)

        painter.end()
        event.accept()
```

A.22 gui/plots/__init__.py

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

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

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

from .classes import (BackgroundPlot, VectorGridPlot,
                      VisualizeTransformationPlot)
from .widgets import (DefineMatrixVisuallyWidget, DefinePolygonWidget,
                      MainViewportWidget, VisualizeTransformationWidget)

__all__ = ['BackgroundPlot', 'DefinePolygonWidget', 'DefineMatrixVisuallyWidget', 'MainViewportWidget',
           'VectorGridPlot', 'VisualizeTransformationPlot', 'VisualizeTransformationWidget']
```

A.23 gui/plots/classes.py

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

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

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

from __future__ import annotations

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

import numpy as np
from PyQt5.QtCore import QPoint, QPointF, QRectF, Qt
```

```
17      from PyQt5.QtGui import (QBrush, QColor, QFont, QMouseEvent, QPainter,
18                               QPainterPath, QPaintEvent, QPen, QPolygonF,
19                               QWheelEvent)
20      from PyQt5.QtWidgets import QWidget
21
22      from lintrans.global_settings import GlobalSettings
23      from lintrans.typing_ import MatrixType, VectorType
24
25
26      class BackgroundPlot(QWidget):
27          """This class provides a background for plotting, as well as setup for a Qt widget.
28
29          This class provides a background (untransformed) plane, and all the backend details
30          for a Qt application, but does not provide useful functionality. To be useful,
31          this class must be subclassed and behaviour must be implemented by the subclass.
32          """
33
34          DEFAULT_GRID_SPACING: int = 85
35          """This is the starting spacing between grid lines (in pixels)."""
36
37          _MINIMUM_GRID_SPACING: int = 5
38          """This is the minimum spacing between grid lines (in pixels)."""
39
40          _COLOUR_BACKGROUND_GRID: QColor = QColor('#808080')
41          """This is the colour of the background grid lines."""
42
43          _COLOUR_BACKGROUND_AXES: QColor = QColor('#000000')
44          """This is the colour of the background axes."""
45
46          _WIDTH_BACKGROUND_GRID: float = 0.3
47          """This is the width of the background grid lines, as a multiple of the :class:`QPainter` line width."""
48
49          _PEN_POLYGON: QPen = QPen(QColor('#000000'), 1.5)
50          """This is the pen used to draw the normal polygon."""
51
52          _BRUSH_SOLID_WHITE: QBrush = QBrush(QColor('#FFFFFF'), Qt.SolidPattern)
53          """This brush is just solid white. Used to draw the insides of circles."""
54
55          def __init__(self, *args, **kwargs):
56              """Create the widget and setup backend stuff for rendering.
57
58              .. note:: ``*args`` and ``**kwargs`` are passed the superclass constructor (:class:`QWidget`).
59              """
60              super().__init__(*args, **kwargs)
61
62              self.setAutoFillBackground(True)
63
64              # Set the background to white
65              palette = self.palette()
66              palette.setColor(self.backgroundRole(), Qt.white)
67              self.setPalette(palette)
68
69              self.grid_spacing = self.DEFAULT_GRID_SPACING
70
71          @property
72          def _canvas_origin(self) -> Tuple[int, int]:
73              """Return the canvas coords of the grid origin.
74
75              The return value is intended to be unpacked and passed to a :meth:`QPainter.drawLine:iiii` call.
76
77              See :meth:`canvas_coords`.
78
79              :returns: The canvas coordinates of the grid origin
80              :rtype: Tuple[int, int]
81              """
82              return self.width() // 2, self.height() // 2
83
84          def _canvas_x(self, x: float) -> int:
85              """Convert an x coordinate from grid coords to canvas coords."""
86              return int(self._canvas_origin[0] + x * self.grid_spacing)
87
88          def _canvas_y(self, y: float) -> int:
89              """Convert a y coordinate from grid coords to canvas coords."""
```

```
90              return int(self._canvas_origin[1] - y * self.grid_spacing)
91
92          def canvas_coords(self, x: float, y: float) -> Tuple[int, int]:
93              """Convert a coordinate from grid coords to canvas coords.
94
95              This method is intended to be used like
96
97              .. code::
98
99                 painter.drawLine(*self.canvas_coords(x1, y1), *self.canvas_coords(x2, y2))
100
101             or like
102
103             .. code::
104
105                painter.drawLine(*self._canvas_origin, *self.canvas_coords(x, y))
106
107             See :attr:`_canvas_origin`.
108
109             :param float x: The x component of the grid coordinate
110             :param float y: The y component of the grid coordinate
111             :returns: The resultant canvas coordinates
112             :rtype: Tuple[int, int]
113             """
114             return self._canvas_x(x), self._canvas_y(y)
115
116         def _grid_corner(self) -> Tuple[float, float]:
117             """Return the grid coords of the top right corner."""
118             return self.width() / (2 * self.grid_spacing), self.height() / (2 * self.grid_spacing)
119
120         def _grid_coords(self, x: int, y: int) -> Tuple[float, float]:
121             """Convert a coordinate from canvas coords to grid coords.
122
123             :param int x: The x component of the canvas coordinate
124             :param int y: The y component of the canvas coordinate
125             :returns: The resultant grid coordinates
126             :rtype: Tuple[float, float]
127             """
128             # We get the maximum grid coords and convert them into canvas coords
129             return (x - self._canvas_origin[0]) / self.grid_spacing, (-y + self._canvas_origin[1]) / self.grid_spacing
130
131         @abstractmethod
132         def paintEvent(self, event: QPaintEvent) -> None:
133             """Handle a :class:`QPaintEvent`.
134
135             .. note:: This method is abstract and must be overridden by all subclasses.
136             """
137
138         def _draw_background(self, painter: QPainter, draw_grid: bool) -> None:
139             """Draw the background grid.
140
141             .. note:: This method is just a utility method for subclasses to use to render the background grid.
142
143             :param QPainter painter: The painter to draw the background with
144             :param bool draw_grid: Whether to draw the grid lines
145             """
146             if draw_grid:
147                 painter.setPen(QPen(self._COLOUR_BACKGROUND_GRID, self._WIDTH_BACKGROUND_GRID))
148
149                 # Draw equally spaced vertical lines, starting in the middle and going out
150                 # We loop up to half of the width. This is because we draw a line on each side in each iteration
151                 for x in range(self.width() // 2 + self.grid_spacing, self.width(), self.grid_spacing):
152                     painter.drawLine(x, 0, x, self.height())
153                     painter.drawLine(self.width() - x, 0, self.width() - x, self.height())
154
155                 # Same with the horizontal lines
156                 for y in range(self.height() // 2 + self.grid_spacing, self.height(), self.grid_spacing):
157                     painter.drawLine(0, y, self.width(), y)
158                     painter.drawLine(0, self.height() - y, self.width(), self.height() - y)
159
160             # Now draw the axes
161             painter.setPen(QPen(self._COLOUR_BACKGROUND_AXES, self._WIDTH_BACKGROUND_GRID))
162             painter.drawLine(self.width() // 2, 0, self.width() // 2, self.height())
```

```
163             painter.drawLine(0, self.height() // 2, self.width(), self.height() // 2)
164
165         def wheelEvent(self, event: QWheelEvent) -> None:
166             """Handle a :class:`QWheelEvent` by zooming in or our of the grid."""
167             # angleDelta() returns a number of units equal to 8 times the number of degrees rotated
168             degrees = event.angleDelta() / 8
169
170             if degrees is not None:
171                 new_spacing = max(1, self.grid_spacing + degrees.y())
172
173                 if new_spacing >= self._MINIMUM_GRID_SPACING:
174                     self.grid_spacing = new_spacing
175
176             event.accept()
177             self.update()
178
179
180     class InteractivePlot(BackgroundPlot):
181         """This class represents an interactive plot, which allows the user to click and/or drag point(s).
182
183         It declares the Qt methods needed for mouse cursor interaction to be abstract,
184         requiring all subclasses to implement these.
185         """
186
187         def _round_to_int_coord(self, point: Tuple[float, float]) -> Tuple[float, float]:
188             """Take a coordinate in grid coords and round it to an integer coordinate if it's within the snapping
    ↪ distance.
189
190             If the point is not close enough, we just return the original point.
191             See :attr:`lintrans.global_settings.GlobalSettingsData.snap_dist`.
192             """
193             x, y = point
194
195             possible_snaps: List[Tuple[int, int]] = [
196                 (floor(x), floor(y)),
197                 (floor(x), ceil(y)),
198                 (ceil(x), floor(y)),
199                 (ceil(x), ceil(y))
200             ]
201
202             snap_distances: List[Tuple[float, Tuple[int, int]]] = [
203                 (dist((x, y), coord), coord)
204                 for coord in possible_snaps
205             ]
206
207             for snap_dist, coord in snap_distances:
208                 if GlobalSettings().get_data().snap_to_int_coords and snap_dist < GlobalSettings().get_data().snap_dist:
209                     x, y = coord
210
211             return x, y
212
213         def _is_within_epsilon(self, cursor_pos: Tuple[float, float], point: Tuple[float, float]) -> bool:
214             """Check if the cursor position (in canvas coords) is within range of the given point."""
215             mx, my = cursor_pos
216             px, py = self.canvas_coords(*point)
217             cursor_epsilon = GlobalSettings().get_data().cursor_epsilon
218             return (abs(px - mx) <= cursor_epsilon and abs(py - my) <= cursor_epsilon)
219
220         @abstractmethod
221         def mousePressEvent(self, event: QMouseEvent) -> None:
222             """Handle the mouse being pressed."""
223
224         @abstractmethod
225         def mouseReleaseEvent(self, event: QMouseEvent) -> None:
226             """Handle the mouse being released."""
227
228         @abstractmethod
229         def mouseMoveEvent(self, event: QMouseEvent) -> None:
230             """Handle the mouse moving on the widget."""
231
232
233     class VectorGridPlot(BackgroundPlot):
```

```
234         """This class represents a background plot, with vectors and their grid drawn on top. It provides utility
    ↪ methods.
235
236         .. note::
237             This is a simple superclass for vectors and is not for visualizing transformations.
238             See :class:`VisualizeTransformationPlot`.
239
240         This class should be subclassed to be used for visualization and matrix definition widgets.
241         All useful behaviour should be implemented by any subclass.
242
243         .. warning:: This class should never be directly instantiated, only subclassed.
244         """
245
246         _COLOUR_I = QColor('#0808d8')
247         """This is the colour of the `i` basis vector and associated transformed grid lines."""
248
249         _COLOUR_J = QColor('#e90000')
250         """This is the colour of the `j` basis vector and associated transformed grid lines."""
251
252         _COLOUR_TEXT = QColor('#000000')
253         """This is the colour of the text."""
254
255         _WIDTH_VECTOR_LINE = 1.8
256         """This is the width of the transformed basis vector lines, as a multiple of the :class:`QPainter` line
    ↪ width."""
257
258         _WIDTH_TRANSFORMED_GRID = 0.8
259         """This is the width of the transformed grid lines, as a multiple of the :class:`QPainter` line width."""
260
261         _ARROWHEAD_LENGTH = 0.15
262         """This is the minimum length (in grid coord size) of the arrowhead parts."""
263
264         _MAX_PARALLEL_LINES = 150
265         """This is the maximum number of parallel transformed grid lines that will be drawn.
266
267         The user can zoom out further, but we will stop drawing grid lines beyond this number.
268         """
269
270         def __init__(self, *args, **kwargs):
271             """Create the widget with ``point_i`` and ``point_j`` attributes.
272
273             .. note:: ``*args`` and ``**kwargs`` are passed to the superclass constructor (:class:`BackgroundPlot`).
274             """
275             super().__init__(*args, **kwargs)
276
277             self.point_i: Tuple[float, float] = (1., 0.)
278             self.point_j: Tuple[float, float] = (0., 1.)
279
280         @property
281         def _matrix(self) -> MatrixType:
282             """Return the assembled matrix of the basis vectors."""
283             return np.array([
284                 [self.point_i[0], self.point_j[0]],
285                 [self.point_i[1], self.point_j[1]]
286             ])
287
288         @property
289         def _det(self) -> float:
290             """Return the determinant of the assembled matrix."""
291             return float(np.linalg.det(self._matrix))
292
293         @property
294         def _eigs(self) -> 'Iterable[Tuple[float, VectorType]]':
295             """Return the eigenvalues and eigenvectors zipped together to be iterated over.
296
297             :rtype: Iterable[Tuple[float, VectorType]]
298             """
299             values, vectors = np.linalg.eig(self._matrix)
300             return zip(values, vectors.T)
301
302         @abstractmethod
303         def paintEvent(self, event: QPaintEvent) -> None:
304             """Handle a :class:`QPaintEvent`."""
```

```
305
306         def _draw_parallel_lines(self, painter: QPainter, vector: Tuple[float, float], point: Tuple[float, float]) ->
            ↪ None:
307             """Draw a set of evenly spaced grid lines parallel to ``vector`` intersecting ``point``.
308
309             :param QPainter painter: The painter to draw the lines with
310             :param vector: The vector to draw the grid lines parallel to
311             :type vector: Tuple[float, float]
312             :param point: The point for the lines to intersect with
313             :type point: Tuple[float, float]
314             """
315             max_x, max_y = self._grid_corner()
316             vector_x, vector_y = vector
317             point_x, point_y = point
318
319             # If the determinant is 0
320             if abs(vector_x * point_y - vector_y * point_x) < 1e-12:
321                 rank = np.linalg.matrix_rank(
322                     np.array([
323                         [vector_x, point_x],
324                         [vector_y, point_y]
325                     ])
326                 )
327
328                 # If the matrix is rank 1, then we can draw the column space line
329                 if rank == 1:
330                     # If the vector does not have a 0 x or y component, then we can just draw the line
331                     if abs(vector_x) > 1e-12 and abs(vector_y) > 1e-12:
332                         self._draw_oblique_line(painter, vector_y / vector_x, 0)
333
334                     # Otherwise, we have to draw lines along the axes
335                     elif abs(vector_x) > 1e-12 and abs(vector_y) < 1e-12:
336                         painter.drawLine(0, self.height() // 2, self.width(), self.height() // 2)
337
338                     elif abs(vector_x) < 1e-12 and abs(vector_y) > 1e-12:
339                         painter.drawLine(self.width() // 2, 0, self.width() // 2, self.height())
340
341                     # If the vector is (0, 0), then don't draw a line for it
342                     else:
343                         return
344
345                 # If the rank is 0, then we don't draw any lines
346                 else:
347                     return
348
349             elif abs(vector_x) < 1e-12 and abs(vector_y) < 1e-12:
350                 # If both components of the vector are practically 0, then we can't render any grid lines
351                 return
352
353             # Draw vertical lines
354             elif abs(vector_x) < 1e-12:
355                 painter.drawLine(self._canvas_x(0), 0, self._canvas_x(0), self.height())
356
357                 for i in range(min(abs(int(max_x / point_x)), self._MAX_PARALLEL_LINES)):
358                     painter.drawLine(
359                         self._canvas_x((i + 1) * point_x),
360                         0,
361                         self._canvas_x((i + 1) * point_x),
362                         self.height()
363                     )
364                     painter.drawLine(
365                         self._canvas_x(-1 * (i + 1) * point_x),
366                         0,
367                         self._canvas_x(-1 * (i + 1) * point_x),
368                         self.height()
369                     )
370
371             # Draw horizontal lines
372             elif abs(vector_y) < 1e-12:
373                 painter.drawLine(0, self._canvas_y(0), self.width(), self._canvas_y(0))
374
375                 for i in range(min(abs(int(max_y / point_y)), self._MAX_PARALLEL_LINES)):
376                     painter.drawLine(
```

```
                    0,
                    self._canvas_y((i + 1) * point_y),
                    self.width(),
                    self._canvas_y((i + 1) * point_y)
                )
                painter.drawLine(
                    0,
                    self._canvas_y(-1 * (i + 1) * point_y),
                    self.width(),
                    self._canvas_y(-1 * (i + 1) * point_y)
                )

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

            self._draw_oblique_line(painter, m, 0)

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

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

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

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

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

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

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

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

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

```
450
451             # If no intersections fit on the canvas
452             if len(points) < 2:
453                 return False
454
455             # If we can, then draw the line
456             else:
457                 painter.drawLine(
458                     *self.canvas_coords(*points[0]),
459                     *self.canvas_coords(*points[1])
460                 )
461                 return True
462
463         def _draw_transformed_grid(self, painter: QPainter) -> None:
464             """Draw the transformed version of the grid, given by the basis vectors.
465
466             .. note:: This method draws the grid, but not the basis vectors. Use :meth:`_draw_basis_vectors` to draw
    ↪ them.
467
468             :param QPainter painter: The painter to draw the grid lines with
469             """
470             # Draw all the parallel lines
471             painter.setPen(QPen(self._COLOUR_I, self._WIDTH_TRANSFORMED_GRID))
472             self._draw_parallel_lines(painter, self.point_i, self.point_j)
473             painter.setPen(QPen(self._COLOUR_J, self._WIDTH_TRANSFORMED_GRID))
474             self._draw_parallel_lines(painter, self.point_j, self.point_i)
475
476         def _draw_arrowhead_away_from_origin(self, painter: QPainter, point: Tuple[float, float]) -> None:
477             """Draw an arrowhead at ``point``, pointing away from the origin.
478
479             :param QPainter painter: The painter to draw the arrowhead with
480             :param point: The point to draw the arrowhead at, given in grid coords
481             :type point: Tuple[float, float]
482             """
483             # This algorithm was adapted from a C# algorithm found at
484             # http://csharphelper.com/blog/2014/12/draw-lines-with-arrowheads-in-c/
485
486             # Get the x and y coords of the point, and then normalize them
487             # We have to normalize them, or else the size of the arrowhead will
488             # scale with the distance of the point from the origin
489             x, y = point
490             vector_length = np.sqrt(x * x + y * y)
491
492             if vector_length < 1e-12:
493                 return
494
495             nx = x / vector_length
496             ny = y / vector_length
497
498             # We choose a length and find the steps in the x and y directions
499             length = min(
500                 self._ARROWHEAD_LENGTH * self.DEFAULT_GRID_SPACING / self.grid_spacing,
501                 vector_length
502             )
503             dx = length * (-nx - ny)
504             dy = length * (nx - ny)
505
506             # Then we just plot those lines
507             painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x + dx, y + dy))
508             painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x - dy, y + dx))
509
510         def _draw_position_vector(self, painter: QPainter, point: Tuple[float, float], colour: QColor) -> None:
511             """Draw a vector from the origin to the given point.
512
513             :param QPainter painter: The painter to draw the position vector with
514             :param point: The tip of the position vector in grid coords
515             :type point: Tuple[float, float]
516             :param QColor colour: The colour to draw the position vector in
517             """
518             painter.setPen(QPen(colour, self._WIDTH_VECTOR_LINE))
519             painter.drawLine(*self._canvas_origin, *self.canvas_coords(*point))
520             self._draw_arrowhead_away_from_origin(painter, point)
521
```

```
522         def _draw_basis_vectors(self, painter: QPainter) -> None:
523             """Draw arrowheads at the tips of the basis vectors.
524 
525             :param QPainter painter: The painter to draw the basis vectors with
526             """
527             self._draw_position_vector(painter, self.point_i, self._COLOUR_I)
528             self._draw_position_vector(painter, self.point_j, self._COLOUR_J)
529 
530         def _draw_basis_vector_labels(self, painter: QPainter) -> None:
531             """Label the basis vectors with `i` and `j`."""
532             font = self.font()
533             font.setItalic(True)
534             font.setStyleHint(QFont.Serif)
535 
536             self._draw_text_at_vector_tip(painter, self.point_i, 'i', font)
537             self._draw_text_at_vector_tip(painter, self.point_j, 'j', font)
538 
539         def _draw_text_at_vector_tip(
540             self,
541             painter: QPainter,
542             point: Tuple[float, float],
543             text: str,
544             font: Optional[QFont] = None
545         ) -> None:
546             """Draw the given text at the point as if it were the tip of a vector, using the custom font if given."""
547             offset = 3
548             top_left: QPoint
549             bottom_right: QPoint
550             alignment_flags: int
551             x, y = point
552 
553             if x >= 0 and y >= 0:  # Q1
554                 top_left = QPoint(self._canvas_x(x) + offset, 0)
555                 bottom_right = QPoint(self.width(), self._canvas_y(y) - offset)
556                 alignment_flags = Qt.AlignLeft | Qt.AlignBottom
557 
558             elif x < 0 and y >= 0:  # Q2
559                 top_left = QPoint(0, 0)
560                 bottom_right = QPoint(self._canvas_x(x) - offset, self._canvas_y(y) - offset)
561                 alignment_flags = Qt.AlignRight | Qt.AlignBottom
562 
563             elif x < 0 and y < 0:  # Q3
564                 top_left = QPoint(0, self._canvas_y(y) + offset)
565                 bottom_right = QPoint(self._canvas_x(x) - offset, self.height())
566                 alignment_flags = Qt.AlignRight | Qt.AlignTop
567 
568             else:  # Q4
569                 top_left = QPoint(self._canvas_x(x) + offset, self._canvas_y(y) + offset)
570                 bottom_right = QPoint(self.width(), self.height())
571                 alignment_flags = Qt.AlignLeft | Qt.AlignTop
572 
573             original_font = painter.font()
574 
575             if font is not None:
576                 painter.setFont(font)
577 
578             painter.setPen(QPen(self._COLOUR_TEXT, 1))
579             painter.drawText(QRectF(top_left, bottom_right), alignment_flags, text)
580 
581             painter.setFont(original_font)
582 
583 
584     class VisualizeTransformationPlot(VectorGridPlot):
585         """This class is a superclass for visualizing transformations. It provides utility methods."""
586 
587         _COLOUR_EIGEN = QColor('#13cf00')
588         """This is the colour of the eigenvectors and eigenlines (the spans of the eigenvectors)."""
589 
590         @abstractmethod
591         def paintEvent(self, event: QPaintEvent) -> None:
592             """Handle a :class:`QPaintEvent`."""
593 
594         def _draw_determinant_parallelogram(self, painter: QPainter) -> None:
```

```
595             """Draw the parallelogram of the determinant of the matrix.
596
597             :param QPainter painter: The painter to draw the parallelogram with
598             """
599             if self._det == 0:
600                 return
601
602             path = QPainterPath()
603             path.moveTo(*self._canvas_origin)
604             path.lineTo(*self.canvas_coords(*self.point_i))
605             path.lineTo(*self.canvas_coords(self.point_i[0] + self.point_j[0], self.point_i[1] + self.point_j[1]))
606             path.lineTo(*self.canvas_coords(*self.point_j))
607
608             color = (16, 235, 253) if self._det > 0 else (253, 34, 16)
609             brush = QBrush(QColor(*color, alpha=128), Qt.SolidPattern)
610
611             painter.fillPath(path, brush)
612
613         def _draw_determinant_text(self, painter: QPainter) -> None:
614             """Write the string value of the determinant in the middle of the parallelogram.
615
616             :param QPainter painter: The painter to draw the determinant text with
617             """
618             painter.setPen(QPen(self._COLOUR_TEXT, self._WIDTH_VECTOR_LINE))
619
620             # We're building a QRect that encloses the determinant parallelogram
621             # Then we can center the text in this QRect
622             coords: List[Tuple[float, float]] = [
623                 (0, 0),
624                 self.point_i,
625                 self.point_j,
626                 (
627                     self.point_i[0] + self.point_j[0],
628                     self.point_i[1] + self.point_j[1]
629                 )
630             ]
631
632             xs = [t[0] for t in coords]
633             ys = [t[1] for t in coords]
634
635             top_left = QPoint(*self.canvas_coords(min(xs), max(ys)))
636             bottom_right = QPoint(*self.canvas_coords(max(xs), min(ys)))
637
638             rect = QRectF(top_left, bottom_right)
639
640             painter.drawText(
641                 rect,
642                 Qt.AlignHCenter | Qt.AlignVCenter,
643                 f'{self._det:.2f}'
644             )
645
646         def _draw_eigenvectors(self, painter: QPainter) -> None:
647             """Draw the eigenvectors of the displayed matrix transformation.
648
649             :param QPainter painter: The painter to draw the eigenvectors with
650             """
651             for value, vector in self._eigs:
652                 x = value * vector[0]
653                 y = value * vector[1]
654
655                 if x.imag != 0 or y.imag != 0:
656                     continue
657
658                 self._draw_position_vector(painter, (x, y), self._COLOUR_EIGEN)
659                 self._draw_text_at_vector_tip(painter, (x, y), f'{value:.2f}')
660
661         def _draw_eigenlines(self, painter: QPainter) -> None:
662             """Draw the eigenlines. These are the invariant lines, or the spans of the eigenvectors.
663
664             :param QPainter painter: The painter to draw the eigenlines with
665             """
666             painter.setPen(QPen(self._COLOUR_EIGEN, self._WIDTH_TRANSFORMED_GRID))
667
```

```
            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)

        def _draw_polygon_from_points(self, painter: QPainter, points: List[Tuple[float, float]]) -> None:
            """Draw a polygon from a given list of points.

            This is a helper method for :meth:`_draw_untransformed_polygon` and :meth:`_draw_transformed_polygon`.
            """
            if len(points) > 2:
                painter.drawPolygon(QPolygonF(
                    [QPointF(*self.canvas_coords(*p)) for p in points]
                ))
            elif len(points) == 2:
                painter.drawLine(
                    *self.canvas_coords(*points[0]),
                    *self.canvas_coords(*points[1])
                )

        def _draw_untransformed_polygon(self, painter: QPainter) -> None:
            """Draw the original untransformed polygon with a dashed line."""
            pen = QPen(self._PEN_POLYGON)
            pen.setDashPattern([4, 4])
            painter.setPen(pen)

            self._draw_polygon_from_points(painter, self.polygon_points)

        def _draw_transformed_polygon(self, painter: QPainter) -> None:
            """Draw the transformed version of the polygon."""
            if len(self.polygon_points) == 0:
                return

            painter.setPen(self._PEN_POLYGON)

            # This transpose trick lets us do one matrix multiplication to transform every point in the polygon
            # I learned this from Phil. Thanks Phil
            self._draw_polygon_from_points(
                painter,
                (self._matrix @ np.array(self.polygon_points).T).T
            )
```

B Testing code

B.1 `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 and functions."""

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]])

    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.2 `backend/test_session.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 functionality of saving and loading sessions."""

from pathlib import Path

from conftest import get_test_wrapper

import lintrans
from lintrans.gui.session import Session
from lintrans.gui.settings import DisplaySettings
from lintrans.matrices.wrapper import MatrixWrapper

```

```

def test_save_and_load(tmp_path: Path, test_wrapper: MatrixWrapper) -> None:
    """Test that sessions save and load and return the same matrix wrapper."""
    points = [(1, 0), (-2, 3), (3.2, -10), (0, 0), (-2, -3), (2, -1.3)]
    session = Session(
        matrix_wrapper=test_wrapper,
        polygon_points=points,
        display_settings=DisplaySettings(),
        input_vector=(2, 3)
    )

    path = str((tmp_path / 'test.lt').absolute())
    session.save_to_file(path)

    loaded_session, version, extra_attrs = Session.load_from_file(path)
    assert loaded_session.matrix_wrapper == get_test_wrapper()
    assert loaded_session.polygon_points == points
    assert loaded_session.display_settings == DisplaySettings()
    assert loaded_session.input_vector == (2, 3)

    assert version == lintrans.__version__
    assert not extra_attrs
```

B.3 backend/matrices/test_parse_and_validate_expression.py

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

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

"""Test the :mod:`matrices.parse` module validation and parsing."""

from typing import List, Tuple

import pytest

from lintrans.matrices.parse import (MatrixParseError, find_sub_expressions,
                                     get_matrix_identifiers,
                                     parse_matrix_expression, strip_whitespace,
                                     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']),
    ('(2.3 A^-1 (A B)^-1 + (B C)^2)', ['2.3A^-1(AB)^-1+(BC)^2']),
    ('A([1 2; 3 4]M^T)^2', ['[1 2;3 4]M^T']),
    ('[1 2; -3 -1]', []),
]


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


expected_stripped_whitespace: List[Tuple[str, str]] = [
    ('[ 1 2 ; 3 4 ]', '[1 2;3 4]'),
    ('[-3.4 6; 1.2 -9 ]', '[-3.4 6;1.2 -9]'),
    ('A    4  [   43      -653.23   ;  32523    -4.3  ]  Z^2', 'A4[43 -653.23;32523 -4.3]Z^2'),
    ('[  1  2; -4   3.64]    [ -5 6; 8.3 2]', '[1 2;-4 3.64][-5 6;8.3 2]')
]

```

```
46
47      def test_strip_whitespace() -> None:
48          """Test the :func:`lintrans.matrices.parse.strip_whitespace` function."""
49          for inp, output in expected_stripped_whitespace:
50              assert strip_whitespace(inp) == output
51
52
53      valid_inputs: List[str] = [
54          'A', 'AB', '3A', '1.2A', '-3.4A', 'A^2', 'A^-1', 'A^{-1}',
55          'A^12', 'A^T', 'A^{5}', 'A^{T}', '4.3A^7', '9.2A^{18}', '0.1A'
56
57          'rot(45)', 'rot(12.5)', '3rot(90)',
58          'rot(135)^3', 'rot(51)^T', 'rot(-34)^-1',
59
60          'A+B', 'A+2B', '4.3A+9B', 'A^2+B^T', '3A^7+0.8B^{16}',
61          'A-B', '3A-4B', '3.2A^3-16.79B^T', '4.752A^{17}-3.32B^{36}',
62          'A-1B', '-A', '-1A', 'A^{2}3.4B', 'A^{-1}2.3B',
63
64          '3A4B', 'A^TB', 'A^{T}B', '4A^6B^3',
65          '2A^{3}4B^5', '4rot(90)^3', 'rot(45)rot(13)',
66          'Arot(90)', 'AB^2', 'A^2B^2', '8.36A^T3.4B^12',
67
68          '3.5A^{4}5.6rot(19.2)^T-B^{-1}4.1C^5',
69
70          '(A)', '(AB)^-1', '2.3(3B^TA)^2', '-3.4(9D^{2}3F^-1)^T+C', '(AB)(C)',
71          '3(rot(34)^-7A)^-1+B', '3A^2B+4A(B+C)^-1D^T-A(C(D+E)B)',
72
73          '[1 2; 3 4]', '4[1 -2;12 5]^3', '[1     -2;         3.1          -4.1365]', 'A[1 -3; 4 5]^-1',
74          'rot(45)[-13.2 9;1.414 0]^2M^T', '([1 2; 3 4])', '3A^2(M-B^T)^{-1}18([13.2 -6.4; -11 0.2]+F)^2'
75      ]
76
77      invalid_inputs: List[str] = [
78          '', 'rot()', 'A^', 'A^1.2', 'A^2 3.4B', 'A^23.4B', 'A^-1 2.3B', 'A^{3.4}', '1,2A', 'ro(12)', '5', '12^2',
79          '^T', '^{12}', '.1A', 'A^{13', 'A^3}', 'A^A', '^2', 'A--B', '--A', '+A', '--1A', 'A--B', 'A--1B',
80          '.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)',
81          '3()^2', '4(your mum)^T', 'rot()', 'rot(10.1.1)', 'rot(--2)', '[]', '[1 2]', '[-1;3]', '[2 3; 5.6]',
82          '1 2; 3 4', '[1 2; 34]', '[1 2 3; 4 5]', '[1 2 3; 4 5 6]', '[;]', '[1; 2 3 4]',
83
84          'This is 100% a valid matrix expression, I swear'
85      ]
86
87
88      @pytest.mark.parametrize('inputs,output', [(valid_inputs, True), (invalid_inputs, False)])
89      def test_validate_matrix_expression(inputs: List[str], output: bool) -> None:
90          """Test the validate_matrix_expression() function."""
91          for inp in inputs:
92              assert validate_matrix_expression(inp) == output
93
94
95      expressions_and_parsed_expressions: List[Tuple[str, MatrixParseList]] = [
96          # Simple expressions
97          ('A', [[('', 'A', '')]]),
98          ('A^2', [[('', 'A', '2')]]),
99          ('A^{2}', [[('', 'A', '2')]]),
100         ('3A', [[('3', 'A', '')]]),
101         ('1.4A^3', [[('1.4', 'A', '3')]]),
102         ('0.1A', [[('0.1', 'A', '')]]),
103         ('0.1A', [[('0.1', 'A', '')]]),
104         ('A^12', [[('', 'A', '12')]]),
105         ('A^234', [[('', 'A', '234')]]),
106
107         # Multiplications
108         ('A 0.1B', [[('', 'A', ''), ('0.1', 'B', '')]]),
109         ('A^2 3B', [[('', 'A', '23'), ('', 'B', '')]]),
110         ('A^{2}3.4B', [[('', 'A', '2'), ('3.4', 'B', '')]]),
111         ('4A^{3} 6B^2', [[('4', 'A', '3'), ('6', 'B', '2')]]),
112         ('4.2A^{T} 6.1B^-1', [[('4.2', 'A', 'T'), ('6.1', 'B', '-1')]]),
113         ('-1.2A^2 rot(45)^2', [[('-1.2', 'A', '2'), ('', 'rot(45)', '2')]]),
114         ('3.2A^T 4.5B^{5} 9.6rot(121.3)', [[('3.2', 'A', 'T'), ('4.5', 'B', '5'), ('9.6', 'rot(121.3)', '')]]),
115         ('-1.18A^{-2} 0.1B^{2} 9rot(-34.6)^-1', [[('-1.18', 'A', '-2'), ('0.1', 'B', '2'), ('9', 'rot(-34.6)', '-1')]]),
116
117         # Additions
118         ('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')]]),
        ('3A^{2}-3B', [[('3', 'A', '2')], [('-3', 'B', '')]]),
        (
            '3MA^{2}-15B^TT',
            [
                [('3', 'M', ''), ('', 'A', '2')],
                [('-15', 'B', 'T'), ('', 'T', '')]
            ]
        ),

        # 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')]]),

        # Anonymous matrices
        ('[1 2; 3 4]', [[('', '[1 2;3 4]', '')]]),
        ('A[-3 4; 16.2 87.93]', [[('', 'A', '')], ('', '[-3 4;16.2 87.93]', '')]]),
        (
            '3A^2(M-[  1 2   ;   5 4 ]^T)^{-1}18([13.2    -6.4;       -11     0.2]+F)^2+Z',
            [
                [('3', 'A', '2'), ('', 'M+-1[1 2;5 4]^{T}', '-1'), ('18', '[13.2 -6.4;-11 0.2]+F', '2')],
                [('', 'Z', '')]
            ]
        ),
        ('[1 2; -3 -1]^-1', [[('', '[1 2;-3 -1]', '-1')]]),
        ('[1 2; -3 -1][-5 6; 8.3 2]', [[('', '[1 2;-3 -1]', ''), ('', '[-5 6;8.3 2]', '')]]),
        (
            '3M[1 2; -3 -1]^{2}-[-5 6; 8.3 2]^TT',
            [
                [('3', 'M', ''), ('', '[1 2;-3 -1]', '2')],
                [('-1', '[-5 6;8.3 2]', 'T'), ('', 'T', '')]
            ]
        ),
        (
            '3M[1 2; -3 -1]^{2}-15[-5 6; 8.3 2]^TT',
            [
                [('3', 'M', ''), ('', '[1 2;-3 -1]', '2')],
                [('-15', '[-5 6;8.3 2]', 'T'), ('', 'T', '')]
            ]
        ),
    ]


    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(strip_whitespace(expression)) == parsed_expression

        for expression in valid_inputs:
            # Assert that it doesn't raise MatrixParseError
            parse_matrix_expression(expression)


    def test_parse_error() -> None:
        """Test that parse_matrix_expression() raises a MatrixParseError."""
        for expression in invalid_inputs:
            with pytest.raises(MatrixParseError):
```

```
            parse_matrix_expression(expression)


def test_get_matrix_identifiers() -> None:
    """Test that matrix identifiers can be properly found."""
    assert get_matrix_identifiers('M^T') == {'M'}
    assert get_matrix_identifiers('ABCDEF') == {'A', 'B', 'C', 'D', 'E', 'F'}
    assert get_matrix_identifiers('AB^{-1}3Crot(45)2A(B^2C^-1)') == {'A', 'B', 'C'}
    assert get_matrix_identifiers('A^{2}3A^-1A^TA') == {'A'}
    assert get_matrix_identifiers('rot(45)(rot(25)rot(20))^2') == set()

    for expression in invalid_inputs:
        with pytest.raises(MatrixParseError):
            get_matrix_identifiers(expression)
```

B.4 backend/matrices/utility/test_rotation_matrices.py

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

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

"""Test 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."""
    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.5 backend/matrices/utility/test_coord_conversion.py

```
# lintrans - The linear transformation visualizer
# Copyright (C) 2022 D. Dyson (DoctorDalek1963)
#
# This program is licensed under GNU GPLv3, available here:
# <https://www.gnu.org/licenses/gpl-3.0.html>

"""Test 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.6 backend/matrices/utility/test_float_utility_functions.py

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

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

"""Test the utility functions for GUI dialog boxes."""

from typing import List, Tuple

import numpy as np
import pytest

from lintrans.matrices.utility import is_valid_float, round_float

valid_floats: List[str] = [
    '0', '1', '3', '-2', '123', '-208', '1.2', '-3.5', '4.252634', '-42362.352325',
    '1e4', '-2.59e3', '4.13e-6', '-5.5244e-12'
]

invalid_floats: List[str] = [
    '', 'pi', 'e', '1.2.3', '1,2', '-', '.', 'None', 'no', 'yes', 'float'
]


@pytest.mark.parametrize('inputs,output', [(valid_floats, True), (invalid_floats, False)])
def test_is_valid_float(inputs: List[str], output: bool) -> None:
    """Test the is_valid_float() function."""
    for inp in inputs:
        assert is_valid_float(inp) == output


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.7 backend/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
import pytest
from conftest import get_test_wrapper
from numpy import linalg as la
from pytest import approx

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


def test_simple_matrix_addition(test_wrapper: MatrixWrapper) -> None:
    """Test simple addition and subtraction of two matrices."""
    # NOTE: We assert that all of these values are not None just to stop mypy complaining
    # These values will never actually be None because they're set in the wrapper() fixture
    # There's probably a better way  do this, because this method is a bit of a bodge, but this works for now
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

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

    assert test_wrapper == get_test_wrapper()


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

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

    assert test_wrapper.evaluate_expression('A2B') == approx(test_wrapper['A'] @ (2 * test_wrapper['B']))
    assert test_wrapper.evaluate_expression('2AB') == approx((2 * test_wrapper['A']) @ test_wrapper['B'])
    assert test_wrapper.evaluate_expression('C3D') == approx(test_wrapper['C'] @ (3 * test_wrapper['D']))
    assert test_wrapper.evaluate_expression('4.2E1.2A') == approx((4.2 * test_wrapper['E']) @ (1.2 *
    ↪  test_wrapper['A']))

    assert test_wrapper == get_test_wrapper()


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

    assert (test_wrapper.evaluate_expression('I') == test_wrapper['I']).all()
    assert (test_wrapper.evaluate_expression('AI') == test_wrapper['A']).all()
```

```
70          assert (test_wrapper.evaluate_expression('IA') == test_wrapper['A']).all()
71          assert (test_wrapper.evaluate_expression('GI') == test_wrapper['G']).all()
72          assert (test_wrapper.evaluate_expression('IG') == test_wrapper['G']).all()
73
74          assert (test_wrapper.evaluate_expression('EID') == test_wrapper['E'] @ test_wrapper['D']).all()
75          assert (test_wrapper.evaluate_expression('IED') == test_wrapper['E'] @ test_wrapper['D']).all()
76          assert (test_wrapper.evaluate_expression('EDI') == test_wrapper['E'] @ test_wrapper['D']).all()
77          assert (test_wrapper.evaluate_expression('IEIDI') == test_wrapper['E'] @ test_wrapper['D']).all()
78          assert (test_wrapper.evaluate_expression('EI^3D') == test_wrapper['E'] @ test_wrapper['D']).all()
79
80          assert test_wrapper == get_test_wrapper()
81
82
83      def test_simple_three_matrix_multiplication(test_wrapper: MatrixWrapper) -> None:
84          """Test simple multiplication of two matrices."""
85          assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
86                 test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
87                 test_wrapper['G'] is not None
88
89          assert (test_wrapper.evaluate_expression('ABC') == test_wrapper['A'] @ test_wrapper['B'] @
            ↪  test_wrapper['C']).all()
90          assert (test_wrapper.evaluate_expression('ACB') == test_wrapper['A'] @ test_wrapper['C'] @
            ↪  test_wrapper['B']).all()
91          assert (test_wrapper.evaluate_expression('BAC') == test_wrapper['B'] @ test_wrapper['A'] @
            ↪  test_wrapper['C']).all()
92          assert (test_wrapper.evaluate_expression('EFG') == test_wrapper['E'] @ test_wrapper['F'] @
            ↪  test_wrapper['G']).all()
93          assert (test_wrapper.evaluate_expression('DAC') == test_wrapper['D'] @ test_wrapper['A'] @
            ↪  test_wrapper['C']).all()
94          assert (test_wrapper.evaluate_expression('GAE') == test_wrapper['G'] @ test_wrapper['A'] @
            ↪  test_wrapper['E']).all()
95          assert (test_wrapper.evaluate_expression('FAG') == test_wrapper['F'] @ test_wrapper['A'] @
            ↪  test_wrapper['G']).all()
96          assert (test_wrapper.evaluate_expression('GAF') == test_wrapper['G'] @ test_wrapper['A'] @
            ↪  test_wrapper['F']).all()
97
98          assert test_wrapper == get_test_wrapper()
99
100
101     def test_matrix_inverses(test_wrapper: MatrixWrapper) -> None:
102         """Test the inverses of single matrices."""
103         assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
104                test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
105                test_wrapper['G'] is not None
106
107         assert (test_wrapper.evaluate_expression('A^{-1}') == la.inv(test_wrapper['A'])).all()
108         assert (test_wrapper.evaluate_expression('B^{-1}') == la.inv(test_wrapper['B'])).all()
109         assert (test_wrapper.evaluate_expression('C^{-1}') == la.inv(test_wrapper['C'])).all()
110         assert (test_wrapper.evaluate_expression('D^{-1}') == la.inv(test_wrapper['D'])).all()
111         assert (test_wrapper.evaluate_expression('E^{-1}') == la.inv(test_wrapper['E'])).all()
112         assert (test_wrapper.evaluate_expression('F^{-1}') == la.inv(test_wrapper['F'])).all()
113         assert (test_wrapper.evaluate_expression('G^{-1}') == la.inv(test_wrapper['G'])).all()
114
115         assert (test_wrapper.evaluate_expression('A^-1') == la.inv(test_wrapper['A'])).all()
116         assert (test_wrapper.evaluate_expression('B^-1') == la.inv(test_wrapper['B'])).all()
117         assert (test_wrapper.evaluate_expression('C^-1') == la.inv(test_wrapper['C'])).all()
118         assert (test_wrapper.evaluate_expression('D^-1') == la.inv(test_wrapper['D'])).all()
119         assert (test_wrapper.evaluate_expression('E^-1') == la.inv(test_wrapper['E'])).all()
120         assert (test_wrapper.evaluate_expression('F^-1') == la.inv(test_wrapper['F'])).all()
121         assert (test_wrapper.evaluate_expression('G^-1') == la.inv(test_wrapper['G'])).all()
122
123         assert test_wrapper == get_test_wrapper()
124
125
126     def test_matrix_powers(test_wrapper: MatrixWrapper) -> None:
127         """Test that matrices can be raised to integer powers."""
128         assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
129                test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
130                test_wrapper['G'] is not None
131
132         assert (test_wrapper.evaluate_expression('A^2') == la.matrix_power(test_wrapper['A'], 2)).all()
133         assert (test_wrapper.evaluate_expression('B^4') == la.matrix_power(test_wrapper['B'], 4)).all()
134         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 \
               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()
```

```
208
209     def test_complicated_expressions(test_wrapper: MatrixWrapper) -> None:
210         """Test evaluation of complicated expressions."""
211         assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
212                test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
213                test_wrapper['G'] is not None
214
215         assert (test_wrapper.evaluate_expression('-3.2A^T 4B^{-1} 6C^{-1} + 8.1D^{2} 3.2E^4') ==
216                 (-3.2 * test_wrapper['A'].T) @ (4 * la.inv(test_wrapper['B'])) @ (6 * la.inv(test_wrapper['C']))
217                 + (8.1 * la.matrix_power(test_wrapper['D'], 2)) @ (3.2 * la.matrix_power(test_wrapper['E'], 4))).all()
218
219         assert (test_wrapper.evaluate_expression('53.6D^{2} 3B^T - 4.9F^{2} 2D + A^3 B^-1') ==
220                 (53.6 * la.matrix_power(test_wrapper['D'], 2)) @ (3 * test_wrapper['B'].T)
221                 - (4.9 * la.matrix_power(test_wrapper['F'], 2)) @ (2 * test_wrapper['D'])
222                 + la.matrix_power(test_wrapper['A'], 3) @ la.inv(test_wrapper['B'])).all()
223
224         assert test_wrapper == get_test_wrapper()
225
226
227     def test_parenthesized_expressions(test_wrapper: MatrixWrapper) -> None:
228         """Test evaluation of parenthesized expressions."""
229         assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
230                test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
231                test_wrapper['G'] is not None
232
233         assert (test_wrapper.evaluate_expression('(A^T)^2') == la.matrix_power(test_wrapper['A'].T, 2)).all()
234         assert (test_wrapper.evaluate_expression('(B^T)^3') == la.matrix_power(test_wrapper['B'].T, 3)).all()
235         assert (test_wrapper.evaluate_expression('(C^T)^4') == la.matrix_power(test_wrapper['C'].T, 4)).all()
236         assert (test_wrapper.evaluate_expression('(D^T)^5') == la.matrix_power(test_wrapper['D'].T, 5)).all()
237         assert (test_wrapper.evaluate_expression('(E^T)^6') == la.matrix_power(test_wrapper['E'].T, 6)).all()
238         assert (test_wrapper.evaluate_expression('(F^T)^7') == la.matrix_power(test_wrapper['F'].T, 7)).all()
239         assert (test_wrapper.evaluate_expression('(G^T)^8') == la.matrix_power(test_wrapper['G'].T, 8)).all()
240
241         assert (test_wrapper.evaluate_expression('(rot(45)^1)^T') == create_rotation_matrix(45).T).all()
242         assert (test_wrapper.evaluate_expression('(rot(45)^2)^T') == la.matrix_power(create_rotation_matrix(45),
            ↪  2).T).all()
243         assert (test_wrapper.evaluate_expression('(rot(45)^3)^T') == la.matrix_power(create_rotation_matrix(45),
            ↪  3).T).all()
244         assert (test_wrapper.evaluate_expression('(rot(45)^4)^T') == la.matrix_power(create_rotation_matrix(45),
            ↪  4).T).all()
245         assert (test_wrapper.evaluate_expression('(rot(45)^5)^T') == la.matrix_power(create_rotation_matrix(45),
            ↪  5).T).all()
246
247         assert (test_wrapper.evaluate_expression('D^3(A+6.2F-0.397G^TE)^-2+A') ==
248                 la.matrix_power(test_wrapper['D'], 3) @ la.matrix_power(
249                     test_wrapper['A'] + 6.2 * test_wrapper['F'] - 0.397 * test_wrapper['G'].T @ test_wrapper['E'],
250                     -2
251                 ) + test_wrapper['A']).all()
252
253         assert (test_wrapper.evaluate_expression('-1.2F^{3}4.9D^T(A^2(B+3E^TF)^-1)^2') ==
254                 -1.2 * la.matrix_power(test_wrapper['F'], 3) @ (4.9 * test_wrapper['D'].T) @
255                 la.matrix_power(
256                     la.matrix_power(test_wrapper['A'], 2) @ la.matrix_power(
257                         test_wrapper['B'] + 3 * test_wrapper['E'].T @ test_wrapper['F'],
258                         -1
259                     ),
260                     2
261                 )).all()
262
263
264     def test_anonymous_matrices(test_wrapper: MatrixWrapper) -> None:
265         """Test that anonymous matrices get evaluated correctly."""
266         assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
267                test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
268                test_wrapper['G'] is not None
269
270         assert (test_wrapper.evaluate_expression('[1 2; -3 -1]') == np.array([[1, 2], [-3, -1]])).all()
271         assert (test_wrapper.evaluate_expression('[1 2; -3 -1][-5 6; 8.3 2]') ==
272                 np.array([[1, 2], [-3, -1]]) @ np.array([[-5, 6], [8.3, 2]])).all()
273         assert (test_wrapper.evaluate_expression('[1 2; -3 -1]^-1') == la.inv(np.array([[1, 2], [-3, -1]]))).all()
274         assert (test_wrapper.evaluate_expression('3A[1 2; -3 -1]^{2}-15[-5 6; 8.3 2]^TB') ==
275                 3 * test_wrapper['A'] @ la.matrix_power(np.array([[1, 2], [-3, -1]]), 2)
276                 - 15 * np.array([[-5, 8.3], [6, 2]]) @ test_wrapper['B']).all()
```

```

def test_value_errors(test_wrapper: MatrixWrapper) -> None:
    """Test that evaluate_expression() raises a ValueError for any malformed input."""
    invalid_expressions = ['', '+', '-', 'This is not a valid expression', '3+4',
                           'A+2', 'A^', '^2', 'A^-', 'At', 'A^t', '3^2']

    for expression in invalid_expressions:
        with pytest.raises(ValueError):
            test_wrapper.evaluate_expression(expression)


def test_linalgerror() -> None:
    """Test that certain expressions raise np.linalg.LinAlgError."""
    matrix_a: MatrixType = np.array([
        [0, 0],
        [0, 0]
    ])

    matrix_b: MatrixType = np.array([
        [1, 2],
        [1, 2]
    ])

    wrapper = MatrixWrapper()
    wrapper['A'] = matrix_a
    wrapper['B'] = matrix_b

    assert (wrapper.evaluate_expression('A') == matrix_a).all()
    assert (wrapper.evaluate_expression('B') == matrix_b).all()

    with pytest.raises(np.linalg.LinAlgError):
        wrapper.evaluate_expression('A^-1')

    with pytest.raises(np.linalg.LinAlgError):
        wrapper.evaluate_expression('B^-1')

    assert (wrapper['A'] == matrix_a).all()
    assert (wrapper['B'] == matrix_b).all()
```

B.8 backend/matrices/matrix_wrapper/test_setting_and_getting.py

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

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

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

from typing import Any, Dict, List

import numpy as np
import pytest
from numpy import linalg as la

from lintrans.matrices import MatrixWrapper
from lintrans.typing_ import MatrixType

valid_matrix_names = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
invalid_matrix_names = ['bad name', '123456', 'Th15 Is an 1nV@l1D n@m3', 'abc', 'a']

test_matrix: MatrixType = np.array([[1, 2], [4, 3]])


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

```

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


    def test_get_name_error(new_wrapper: MatrixWrapper) -> None:
        """Test that MatrixWrapper().__getitem__() raises a NameError if called with an invalid name."""
        for name in invalid_matrix_names:
            with pytest.raises(NameError):
                _ = new_wrapper[name]


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

            new_wrapper[name] = None
            assert new_wrapper[name] is None


    def test_set_expression(test_wrapper: MatrixWrapper) -> None:
        """Test that MatrixWrapper.__setitem__() can accept a valid expression."""
        test_wrapper['N'] = 'A^2'
        test_wrapper['O'] = 'BA+2C'
        test_wrapper['P'] = 'E^T'
        test_wrapper['Q'] = 'C^-1B'
        test_wrapper['R'] = 'A^{2}3B'
        test_wrapper['S'] = 'N^-1'
        test_wrapper['T'] = 'PQP^-1'

        with pytest.raises(TypeError):
            test_wrapper['U'] = 'A+1'

        with pytest.raises(TypeError):
            test_wrapper['V'] = 'K'

        with pytest.raises(TypeError):
            test_wrapper['W'] = 'L^2'

        with pytest.raises(TypeError):
            test_wrapper['X'] = 'M^-1'

        with pytest.raises(TypeError):
            test_wrapper['Y'] = 'A^2B+C^'


    def test_simple_dynamic_evaluation(test_wrapper: MatrixWrapper) -> None:
        """Test that expression-defined matrices are evaluated dynamically."""
        test_wrapper['N'] = 'A^2'
        test_wrapper['O'] = '4B'
        test_wrapper['P'] = 'A+C'

        assert (test_wrapper['N'] == test_wrapper.evaluate_expression('A^2')).all()
        assert (test_wrapper['O'] == test_wrapper.evaluate_expression('4B')).all()
        assert (test_wrapper['P'] == test_wrapper.evaluate_expression('A+C')).all()

        assert (test_wrapper.evaluate_expression('N^2 + 3O') ==
                la.matrix_power(test_wrapper.evaluate_expression('A^2'), 2) +
                3 * test_wrapper.evaluate_expression('4B')
                ).all()
        assert (test_wrapper.evaluate_expression('P^-1 - 3NO^2') ==
                la.inv(test_wrapper.evaluate_expression('A+C')) -
                (3 * test_wrapper.evaluate_expression('A^2')) @
                la.matrix_power(test_wrapper.evaluate_expression('4B'), 2)
                ).all()

        test_wrapper['A'] = np.array([
            [19, -21.5],
            [84, 96.572]
        ])
        test_wrapper['B'] = np.array([
            [-0.993, 2.52],
            [1e10, 0]
```

```
        ])
        test_wrapper['C'] = np.array([
            [0, 19512],
            [1.414, 19]
        ])

        assert (test_wrapper['N'] == test_wrapper.evaluate_expression('A^2')).all()
        assert (test_wrapper['O'] == test_wrapper.evaluate_expression('4B')).all()
        assert (test_wrapper['P'] == test_wrapper.evaluate_expression('A+C')).all()

        assert (test_wrapper.evaluate_expression('N^2 + 3O') ==
                la.matrix_power(test_wrapper.evaluate_expression('A^2'), 2) +
                3 * test_wrapper.evaluate_expression('4B')
                ).all()
        assert (test_wrapper.evaluate_expression('P^-1 - 3NO^2') ==
                la.inv(test_wrapper.evaluate_expression('A+C')) -
                (3 * test_wrapper.evaluate_expression('A^2')) @
                la.matrix_power(test_wrapper.evaluate_expression('4B'), 2)
                ).all()


    def test_recursive_dynamic_evaluation(test_wrapper: MatrixWrapper) -> None:
        """Test that dynamic evaluation works recursively."""
        test_wrapper['N'] = 'A^2'
        test_wrapper['O'] = '4B'
        test_wrapper['P'] = 'A+C'

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

        assert test_wrapper['Q'] == pytest.approx(test_wrapper.evaluate_expression('A^-2'))
        assert test_wrapper['R'] == pytest.approx(test_wrapper.evaluate_expression('A + C - 16B'))
        assert test_wrapper['S'] == pytest.approx(test_wrapper.evaluate_expression('A^{2}4BA + A^{2}4BC'))


    def test_self_referential_expressions(test_wrapper: MatrixWrapper) -> None:
        """Test that self-referential expressions raise an error."""
        expressions: Dict[str, str] = {
            'A': 'A^2',
            'B': 'A(C^-1A^T)+rot(45)B',
            'C': '2Brot(1482.536)(A^-1D^{2}4CE)^3F'
        }

        for name, expression in expressions.items():
            with pytest.raises(ValueError):
                test_wrapper[name] = expression

        test_wrapper['B'] = '3A^2'
        test_wrapper['C'] = 'ABBA'
        with pytest.raises(ValueError):
            test_wrapper['A'] = 'C^-1'

        test_wrapper['E'] = 'rot(45)B^-1+C^T'
        test_wrapper['F'] = 'EBDBIC'
        test_wrapper['D'] = 'E'
        with pytest.raises(ValueError):
            test_wrapper['D'] = 'F'


    def test_get_matrix_dependencies(test_wrapper: MatrixWrapper) -> None:
        """Test MatrixWrapper's get_matrix_dependencies() and get_expression_dependencies() methods."""
        test_wrapper['N'] = 'A^2'
        test_wrapper['O'] = '4B'
        test_wrapper['P'] = 'A+C'
        test_wrapper['Q'] = 'N^-1'
        test_wrapper['R'] = 'P-4O'
        test_wrapper['S'] = 'NOP'

        assert test_wrapper.get_matrix_dependencies('A') == set()
        assert test_wrapper.get_matrix_dependencies('B') == set()
        assert test_wrapper.get_matrix_dependencies('C') == set()
        assert test_wrapper.get_matrix_dependencies('D') == set()
```

```
        assert test_wrapper.get_matrix_dependencies('E') == set()
        assert test_wrapper.get_matrix_dependencies('F') == set()
        assert test_wrapper.get_matrix_dependencies('G') == set()

        assert test_wrapper.get_matrix_dependencies('N') == {'A'}
        assert test_wrapper.get_matrix_dependencies('O') == {'B'}
        assert test_wrapper.get_matrix_dependencies('P') == {'A', 'C'}
        assert test_wrapper.get_matrix_dependencies('Q') == {'A', 'N'}
        assert test_wrapper.get_matrix_dependencies('R') == {'A', 'B', 'C', 'O', 'P'}
        assert test_wrapper.get_matrix_dependencies('S') == {'A', 'B', 'C', 'N', 'O', 'P'}

        assert test_wrapper.get_expression_dependencies('ABC') == set()
        assert test_wrapper.get_expression_dependencies('NOB') == {'A', 'B'}
        assert test_wrapper.get_expression_dependencies('N^2O^Trot(90)B^-1') == {'A', 'B'}
        assert test_wrapper.get_expression_dependencies('NOP') == {'A', 'B', 'C'}
        assert test_wrapper.get_expression_dependencies('NOPQ') == {'A', 'B', 'C', 'N'}
        assert test_wrapper.get_expression_dependencies('NOPQR') == {'A', 'B', 'C', 'N', 'O', 'P'}
        assert test_wrapper.get_expression_dependencies('NOPQRS') == {'A', 'B', 'C', 'N', 'O', 'P'}


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


    def test_set_name_error(new_wrapper: MatrixWrapper) -> None:
        """Test that MatrixWrapper().__setitem__() raises a NameError when trying to assign to an invalid name."""
        for name in invalid_matrix_names:
            with pytest.raises(NameError):
                new_wrapper[name] = test_matrix


    def test_set_type_error(new_wrapper: MatrixWrapper) -> None:
        """Test that MatrixWrapper().__setitem__() raises a TypeError when trying to set a non-matrix."""
        invalid_values: List[Any] = [
                                    12,
                                    [1, 2, 3, 4, 5],
                                    [[1, 2], [3, 4]],
                                    True,
                                    24.3222,
                                    'This is totally a matrix, I swear',
                                    MatrixWrapper,
                                    MatrixWrapper(),
                                    np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
                                    np.eye(100)
                                    ]

        for value in invalid_values:
            with pytest.raises(TypeError):
                new_wrapper['M'] = value


    def test_get_expression(test_wrapper: MatrixWrapper) -> None:
        """Test the get_expression method of the MatrixWrapper class."""
        test_wrapper['N'] = 'A^2'
        test_wrapper['O'] = '4B'
        test_wrapper['P'] = 'A+C'

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

        assert test_wrapper.get_expression('A') is None
        assert test_wrapper.get_expression('B') is None
        assert test_wrapper.get_expression('C') is None
        assert test_wrapper.get_expression('D') is None
        assert test_wrapper.get_expression('E') is None
        assert test_wrapper.get_expression('F') is None
        assert test_wrapper.get_expression('G') is None

        assert test_wrapper.get_expression('N') == 'A^2'
        assert test_wrapper.get_expression('O') == '4B'
```

```
248         assert test_wrapper.get_expression('P') == 'A+C'
249
250         assert test_wrapper.get_expression('Q') == 'N^-1'
251         assert test_wrapper.get_expression('R') == 'P-4O'
252         assert test_wrapper.get_expression('S') == 'NOP'
```