lintrans

by D. Dyson

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

Contents

1 Analysis

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

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

1.1 Computational Approach

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

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

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

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

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

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

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

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

1.2 Stakeholders

Stakeholders for my app include A Level Further Maths students and teachers, who learn and teach linear transformations respectively. They will be able to provide useful input as to what they would like to see in the app, and they can provide feedback on what they like and what I can add or improve. I already know from experience that linear transformations are tricky to visualize and a computer-based visualization would be useful. My stakeholders agreed with this. 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'[8] and allows the user to drag an input vector

around the plane and see the corresponding output vector, transformed by a matrix that the user can define, although this definition is finicky since it involves sliders rather than keyboard input.

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

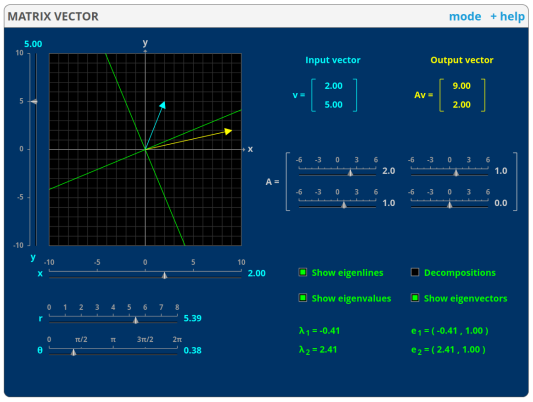


Figure 1.1: The MIT 'Matrix Vector' Mathlet

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

1.3.2 Linear Transformation Visualizer

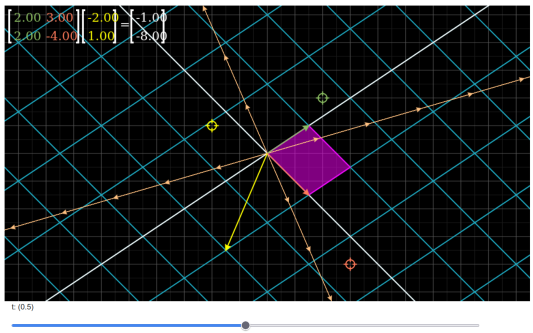


Figure 1.2: 'Linear Transformation Visualizer' halfway through an animation

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

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

1.3.3 Desmos app

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

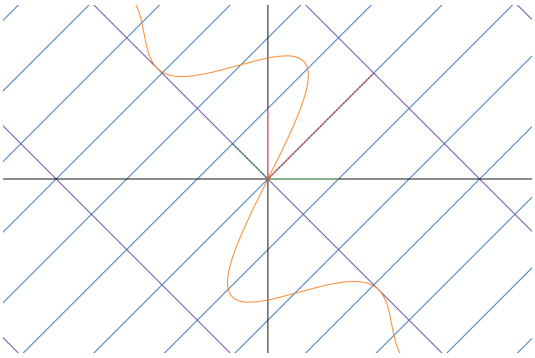


Figure 1.3: The Desmos app halfway through an animation, rendering $f(x) = \frac{\sin^2 x}{x}$ in orange

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

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

1.3.4 Visualizing Linear Transformations

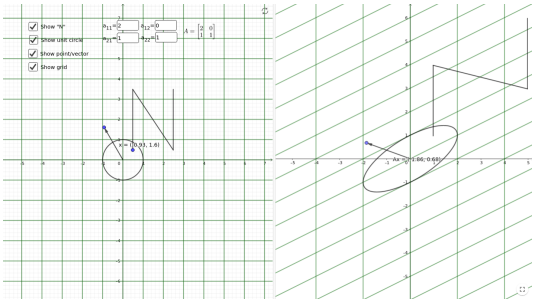


Figure 1.4: The GeoGebra applet rendering its default matrix

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

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

1.4 Essential features

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

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

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

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

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

1.5 Limitations

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

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

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

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

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

1.6 Hardware and software requirements

1.6.1 Hardware

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

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

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

1.6.2 Software

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

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

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

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

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

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

1.7 Success criteria

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

Additionally, the app must fulfil some basic requirements:

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

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

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

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

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

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

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

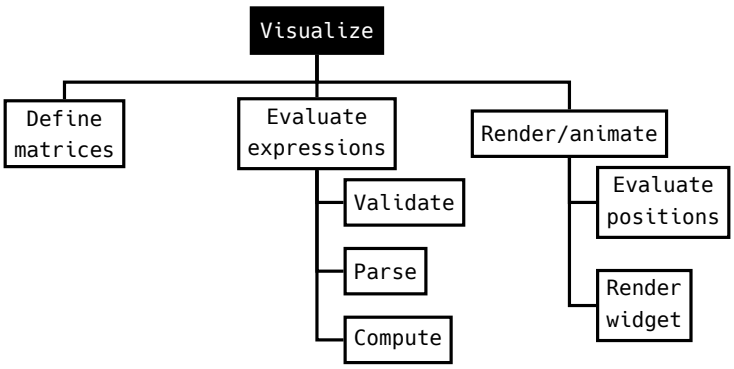
Transforming polygons is the lowest priority item on this list and will likely be implemented last, but it would definitely benefit learning. I wouldn't consider it necessary for success, but it would be very good to include, and it's certainly a feature that I want to have.

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

2 Design

2.1 Problem decomposition

I have decomposed the problem of visualization as follows:



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

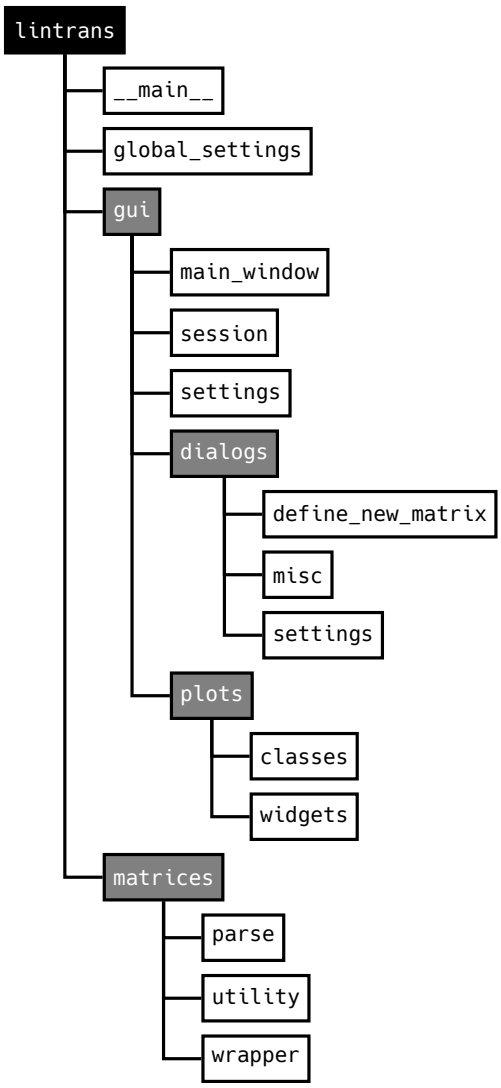
Evaluating expressions is another key part of the app and can be further broken down into validating, parsing, and computing the value. Validating an expression simply consists of checking that it adheres to a set of syntax rules for matrix expressions, and that it only contains matrices which have already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

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

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

2.2 Structure of the solution

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. My pseudocode is actually Python, purely to allow for syntax highlighting.

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

```
import numpy as np

def create_rotation_matrix(angle: float, *, degrees: bool = True) -> MatrixType:
    """Create a matrix representing a rotation (anticlockwise) by the given angle."""
    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)]
    ])
```

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:
    def __init__(self):
        # This dictionary maps all letters of the alphabet to an optional matrix
        self._matrices: Dict[str, Optional[Union[MatrixType, str]]] = {
            'A': None, 'B': None, 'C': None, 'D': None,
```

```
            'E': None, 'F': None, 'G': None, 'H': None,
            'I': np.eye(2),  # I is always defined as the identity matrix
            'J': None, 'K': None, 'L': None, 'M': None,
            'N': None, 'O': None, 'P': None, 'Q': None,
            'R': None, 'S': None, 'T': None, 'U': None,
            'V': None, 'W': None, 'X': None, 'Y': None,
            'Z': None
        }

    def __getitem__(self, name: str) -> Optional[MatrixType]:
        """Get the matrix with the given name.

        If it is a simple name, it will just be fetched from the dictionary. If the name is ``rot(x)``, with
        a given angle in degrees, then we return a new matrix representing a rotation by that angle.

        Using ``__getitem__`` here allows for syntax like ``wrapper['A']`` as if it was a dictionary.
        """
        # Return a new rotation matrix
        if (match := re.match(r'^rot\((-?\d*\.?\d*)\)$', name)) is not None:
            return create_rotation_matrix(float(match.group(1)))

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

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

        return matrix

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

        If ``new_matrix`` is None, then that effectively unsets the matrix name.

        Using ``__getitem__`` here allows for syntax like ``wrapper['A'] = matrix`` as if it was a dictionary.
        """
        if not (name in self._matrices and name != 'I'):
            raise NameError('Matrix name is illegal')

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

        if 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]])
```

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[25], which makes red and green look incredibly similar. I will use blue and red for my basis vectors. These colours are easy to distinguish for people with deuteranopia and protanopia - the two most common forms of colour blindness. Tritanopia makes it harder to distinguish blue and yellow, but my colour scheme is still be accessible for people with tritanopia, as red and blue are very distinct in this form of colour blindness.

I will probably use green for the eigenvectors and eigenlines, which will be hard to distinguish from the red basis vector for people with red-green colour blindness, but I think that the basis vectors and eigenvectors/eigenlines will look physically different enough from each other that the colour shouldn't be too much of a problem. Additionally, I will use a tool called Color Oracle[11] to make sure that my app is accessible to people with different forms of colour blindness[6].

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

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

I will also have hotkeys for everything that can have hotkeys - buttons, checkboxes, etc. This makes my life easier, since I'm used to having hotkeys for everything, and thus makes the app faster to test because I don't need to click everything. This also makes things easier for other people like me, who prefer to stay at the keyboard and not use the mouse. Obviously a mouse will be required for things

[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

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 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.

[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()`

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[15] to allow indexing into a `MatrixWrapper` object like `wrapper['M']`. This simplifies using the class.

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

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

import numpy as np

from lintrans.typing import MatrixType


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

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

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

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

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

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

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

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

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

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

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

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

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

"""Test the MatrixWrapper class."""

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

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


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


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

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


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


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


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


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

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

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

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

3.1.2 Rudimentary parsing and evaluating

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

I then implemented several tests for this parsing.

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

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

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


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

    root_two_over_two = np.sqrt(2) / 2

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

    return wrapper


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

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

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


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

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


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

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

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

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

3.1.3 Simple matrix expression validation

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

```
# 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.

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

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

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

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

"""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

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

to the new version:

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

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

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

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

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

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

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

This method passes all the unit tests, as expected.

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

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

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

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

3.2 Initial GUI

3.2.1 First basic GUI

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

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

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

import sys

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

from lintrans.matrices import MatrixWrapper


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

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

        self.matrix_wrapper = MatrixWrapper()

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

        # === Create widgets

        # Left layout: the plot and input box

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

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

        # Right layout: all the buttons

        # Misc buttons

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

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

        # Define new matrix buttons

```

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

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

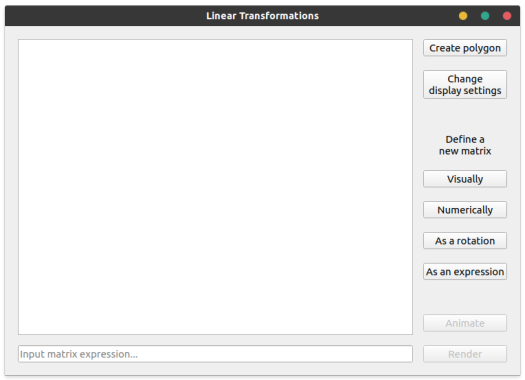


Figure 3.1: The first version of the GUI

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

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

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

3.2.2 Numerical definition dialog

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

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

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

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

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

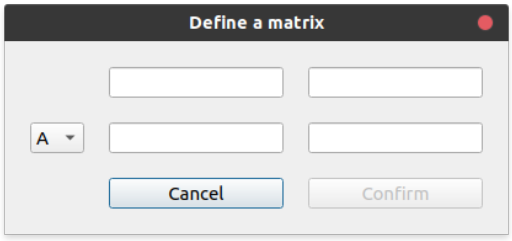


Figure 3.2: The first version of the numerical definition dialog

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

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

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

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

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

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

3.2.3 More definition dialogs

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

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

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

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

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

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

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

I then added the class for the rotation definition dialog.

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

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

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

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

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

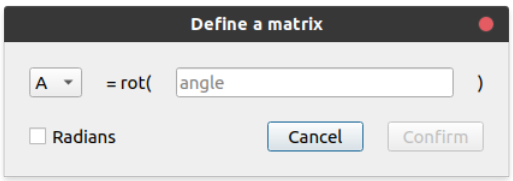


Figure 3.3: The first version of the rotation definition dialog

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

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

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

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

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

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

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

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

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

3.3 Visualizing matrices

3.3.1 Asking strangers on the internet for help

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

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

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

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

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

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

3.3.2 Creating the plots package

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

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

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

from __future__ import annotations

from PyQt5.QtCore import Qt
```

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


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

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

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

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

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

        self.setAutoFillBackground(True)

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

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

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

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

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

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

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

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

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

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

        # Now draw the axes
```

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

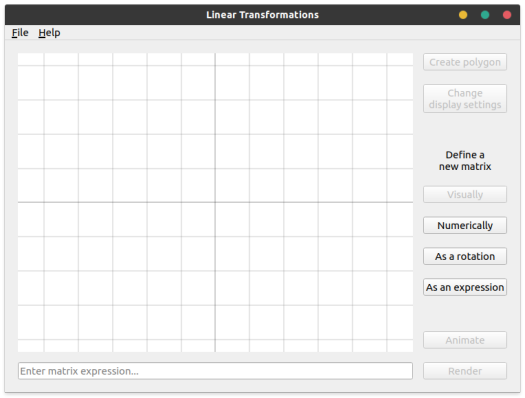


Figure 3.4: The GUI with background axes

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

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

3.3.3 Implementing basis vectors

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

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

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

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

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

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

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

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

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

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

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

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

Testing this new code shows that it works well.

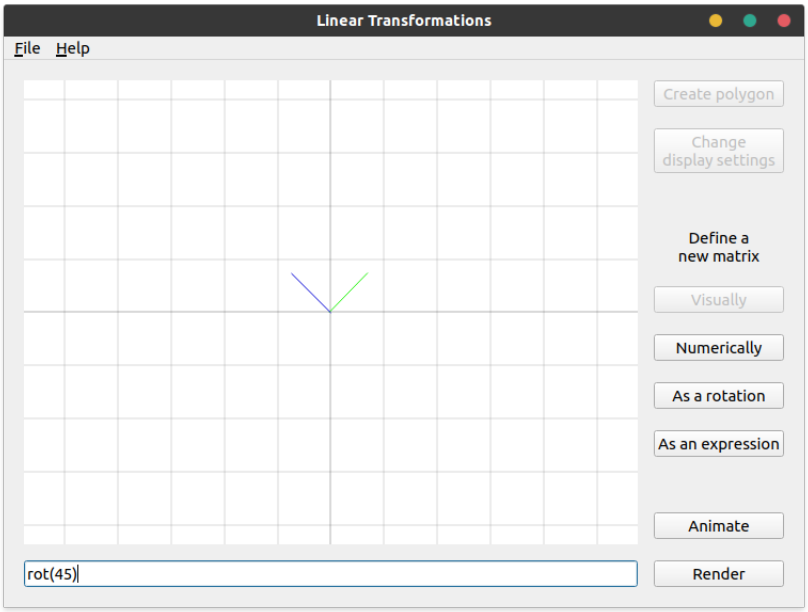


Figure 3.5: Basis vectors drawn for a 45° rotation

3.3.4 Drawing the transformed grid

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

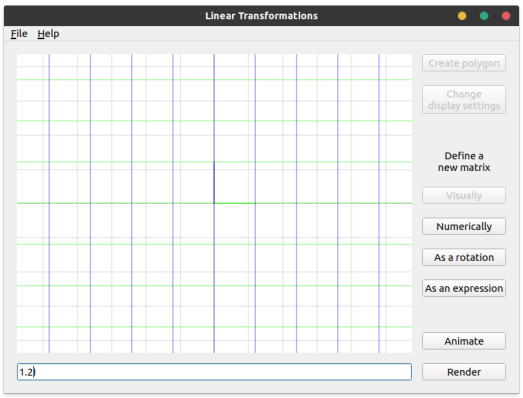


Figure 3.6: Parallel lines being drawn for matrix $1.2\mathbf{I}$

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

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

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

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

This code checks if x or y is zero[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.

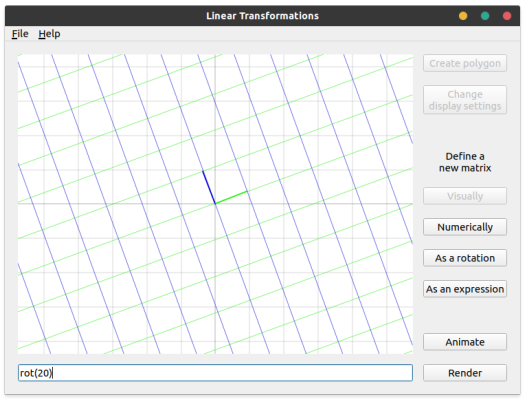


Figure 3.7: An example of a 20° rotation

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 $\left(\begin{smallmatrix} 12 & 4 \\ -2 & 3 \end{smallmatrix}\right)$, because it's just not very good.

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

3.3.5 Implementing animation

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

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

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

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

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

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

3.3.6 Preserving determinants

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

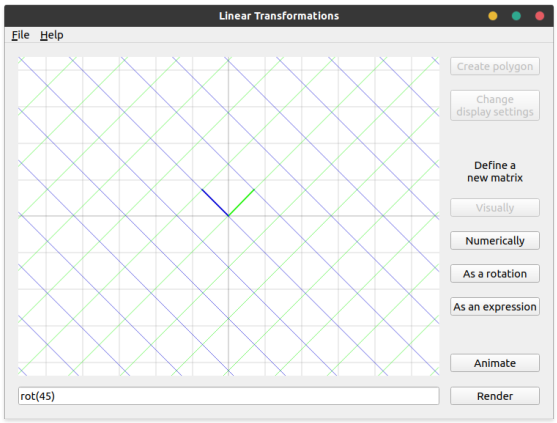


Figure 3.8: What we would expect halfway through a 90° rotation

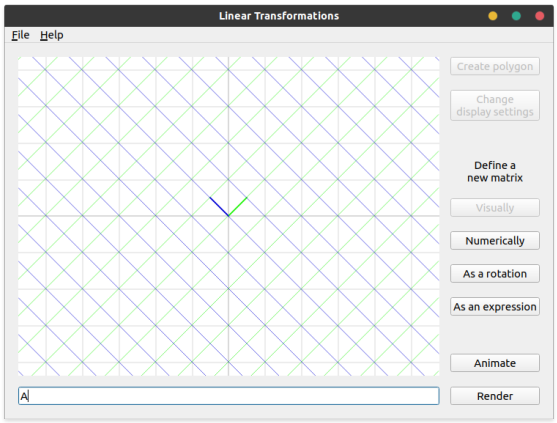


Figure 3.9: What we actually get halfway through a 90° rotation

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

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

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

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

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

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

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

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

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

3.4 Improving the GUI

3.4.1 Fixing rendering

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

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

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

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

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

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

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

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

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

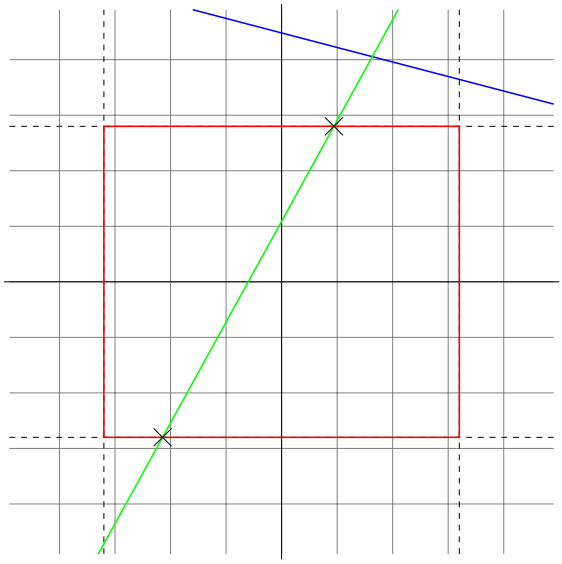


Figure 3.10: Two example lines and the viewport box

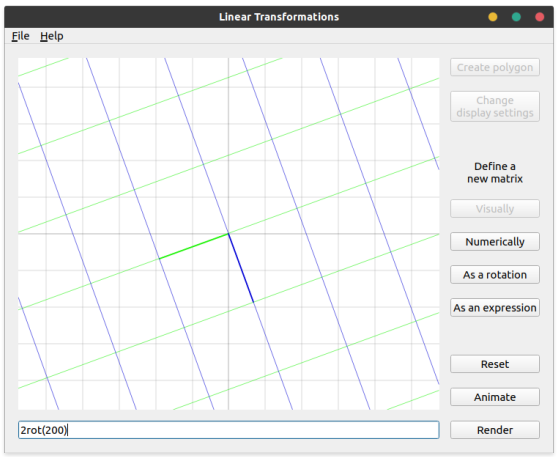


Figure 3.11: A demonstration of the new oblique lines system.

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

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

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

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

3.4.2 Adding vector arrowheads

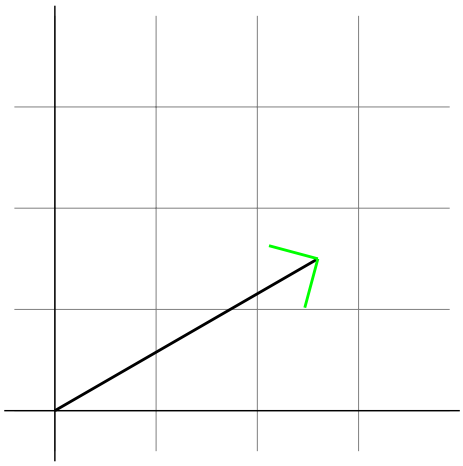


Figure 3.12: An example of a vector with the arrowheads highlighted in green

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

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

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

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

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

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

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

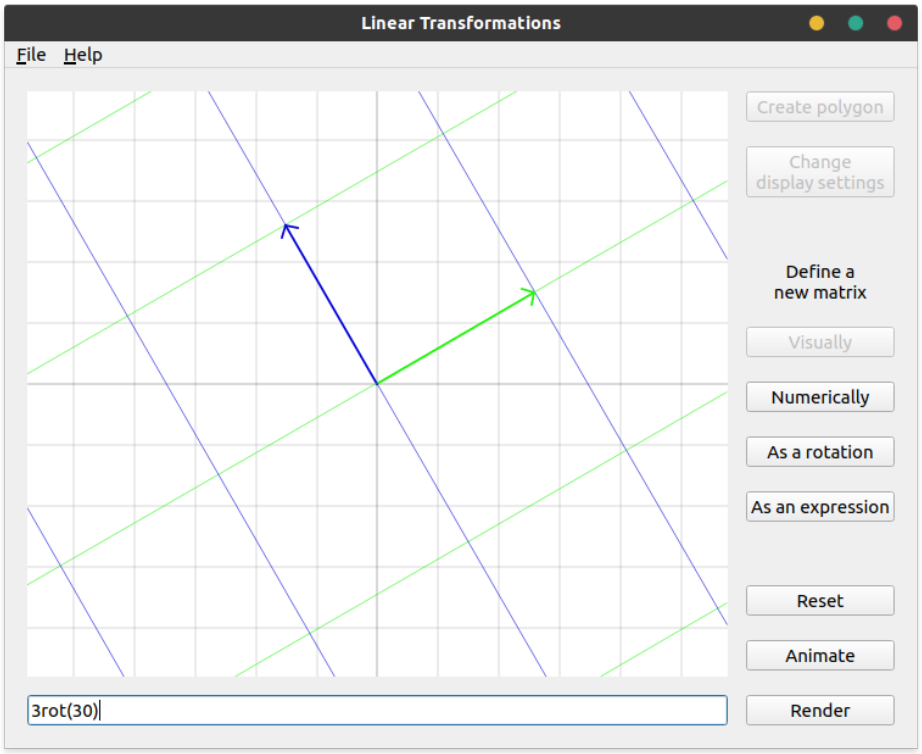


Figure 3.13: An example of the i and j vectors with arrowheads

3.4.3 Implementing zoom

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

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

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

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

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

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

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

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

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

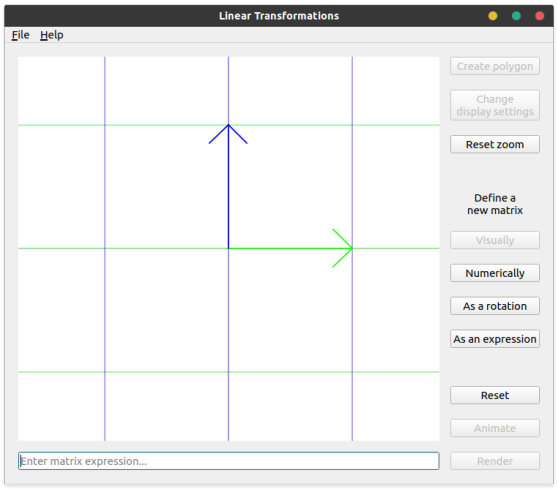


Figure 3.14: The GUI zoomed in a bit

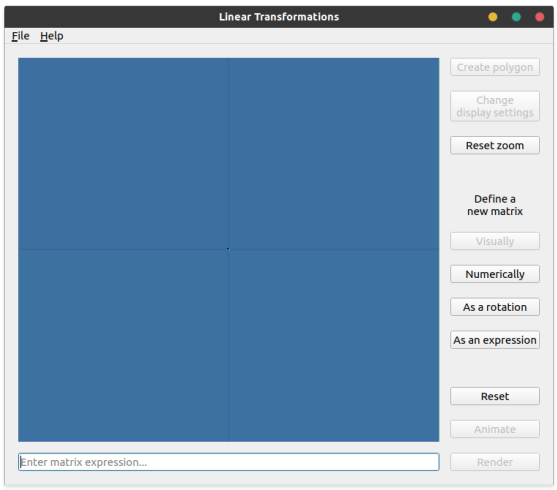


Figure 3.15: The GUI zoomed out as far as possible

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

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

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

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

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

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

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

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

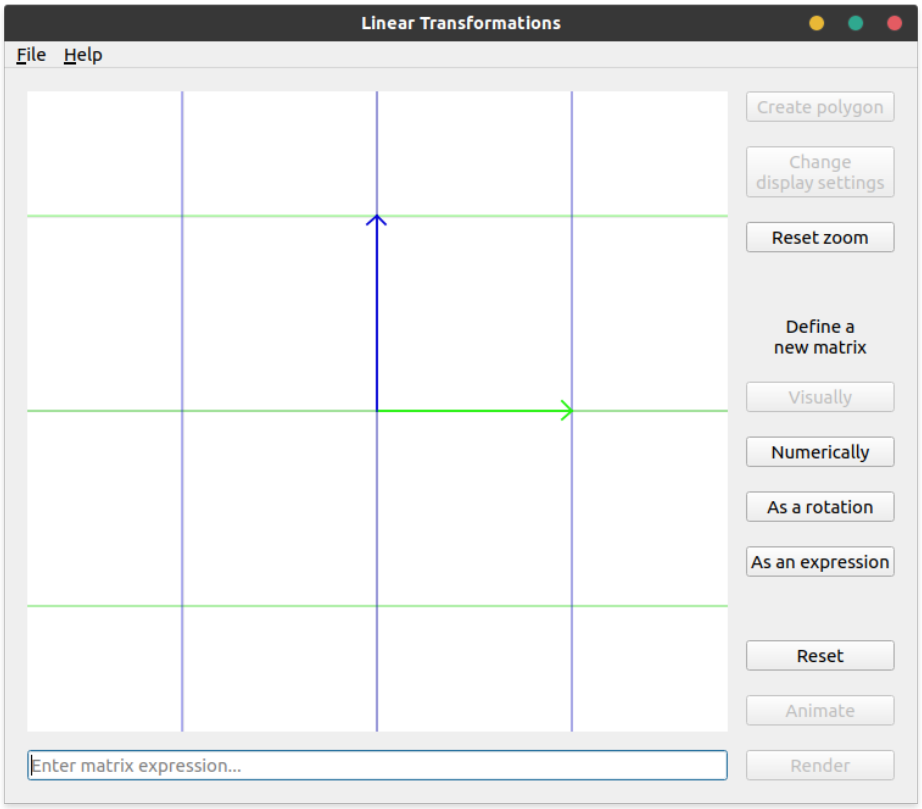


Figure 3.16: The arrowheads adjusted for zoom level

3.4.4 Animation blocks zooming

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

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

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

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

3.4.5 Rank 1 transformations

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

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

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

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

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

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

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

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

3.4.6 Matrices that are too big

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

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

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

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

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

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

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

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

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

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

3.4.7 Creating the `DefineVisuallyDialog`

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

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

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

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

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

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

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

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

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

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

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

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

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

3.4.8 Fixing a division by zero bug

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

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

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

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

3.4.9 Implementing transitional animation

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

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

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

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

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

In code, that looks like this:

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

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

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

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

3.4.10 Allowing for sequential animation with commas

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

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

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

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

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

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

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

Compare the old code to the new code:

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

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

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

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

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

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

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

3.5 Adding display settings

3.5.1 Creating the dataclass

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

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

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

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

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

from dataclasses import dataclass


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

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

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

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

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

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

the determinant to be smoothed.

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

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

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

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

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

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

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

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

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

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

            else:
                matrix_to_render = matrix_a

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

            self.plot.visualize_matrix_transformation(matrix_to_render)

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

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, 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.

```
# e4d3aa7eab70daecd16814972a281745110d64a5
# src/lintrans/gui/dialogs/settings.py

"""This module provides dialogs to edit settings within the app."""

from __future__ import annotations

import abc
import copy

from PyQt5 import QtWidgets
from PyQt5.QtCore import Qt
from PyQt5.QtGui import 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('Cancel this definition<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:
```

```
62          """Confirm the settings chosen in the dialog."""
63
64
65  class DisplaySettingsDialog(SettingsDialog):
66      """The dialog to allow the user to edit the display settings."""
67
68      def __init__(self, display_settings: DisplaySettings, *args, **kwargs):
69          """Create the widgets and layout of the dialog.
70
71          :param DisplaySettings display_settings: The :class:`lintrans.gui.settings.DisplaySettings` object to mutate
72          """
73          super().__init__(*args, **kwargs)
74
75          self.display_settings = display_settings
76          self.setWindowTitle('Change display settings')
77
78          # === 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         # === Arrange the widgets
101
102         self.vlay_options.addWidget(self.label_animations)
103         self.vlay_options.addWidget(self.checkbox_animate_determinant)
104         self.vlay_options.addWidget(self.checkbox_applicative_animation)
105
106         # Finally, we load the current settings
107         self.load_settings()
108
109     def load_settings(self) -> None:
110         """Load the current display settings into the widgets."""
111         self.checkbox_animate_determinant.setChecked(self.display_settings.animate_determinant)
112         self.checkbox_applicative_animation.setChecked(self.display_settings.applicative_animation)
113
114     def confirm_settings(self) -> None:
115         """Build a :class:`lintrans.gui.settings.DisplaySettings` object and assign it."""
116         self.display_settings.animate_determinant = self.checkbox_animate_determinant.isChecked()
117         self.display_settings.applicative_animation = self.checkbox_applicative_animation.isChecked()
118
119         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.

```
# e4d3aa7eab70daecd16814972a281745110d64a5
# src/lintrans/gui/main_window.py

425     def dialog_change_display_settings(self) -> None:
426         """Open the dialog to change the display settings."""
427         dialog = DisplaySettingsDialog(self.display_settings, self)
428         dialog.open()
```

```
429             dialog.finished.connect(lambda: self._assign_display_settings(dialog.display_settings))
430
431         def _assign_display_settings(self, display_settings: DisplaySettings) -> None:
432             """Assign a new value to ``self.display_settings``."""
433             self.display_settings = display_settings
```

The `dialog.finished` signal on line 429 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.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

A Project code

A.1 `crash_reporting.py`

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

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

"""This module provides functions to report crashes and log them.

The only functions you should be calling directly are :func:`set_excepthook`
and :func:`set_signal_handler` to setup handlers for unhandled exceptions
and unhandled operating system signals respectively.
"""

from __future__ import annotations

import os
import platform
import signal
import sys
from datetime import datetime
from signal import SIGABRT, SIGFPE, SIGILL, SIGSEGV, SIGTERM
from textwrap import indent
from types import FrameType, TracebackType
from typing import NoReturn, Type

from PyQt5.QtCore import PYQT_VERSION_STR, QT_VERSION_STR
from PyQt5.QtWidgets import QApplication

import lintrans
from lintrans.typing_ import is_matrix_type

from .global_settings import GlobalSettings
from .gui.main_window import LintransMainWindow


def _get_datetime_string() -> str:
    """Get the date and time as a string with a space in the middle."""
    return datetime.now().strftime('%Y-%m-%d %H:%M:%S')


def _get_main_window() -> LintransMainWindow:
    """Return the only instance of :class:`~lintrans.gui.main_window.LintransMainWindow`.

    :raises RuntimeError: If there is not exactly 1 instance of
    ↪  :class:`~lintrans.gui.main_window.LintransMainWindow`
    """
    widgets = [
        x for x in QApplication.topLevelWidgets()
        if isinstance(x, LintransMainWindow)
    ]

    if len(widgets) != 1:
        raise RuntimeError(f'Expected 1 widget of type LintransMainWindow but found {len(widgets)}')

    return widgets[0]


def _get_system_info() -> str:
    """Return a string of all the system we could gather."""
    info = 'SYSTEM INFO:\n'

    info += f'  lintrans: {lintrans.__version__}\n'
    info += f'  Python: {platform.python_version()}\n'
    info += f'  Qt5: {QT_VERSION_STR}\n'
    info += f'  PyQt5: {PYQT_VERSION_STR}\n'
    info += f'  Platform: {platform.platform()}\n'

```

```
        info += '\n'
        return info


    def _get_error_origin(
        *,
        exc_type: Type[BaseException] | None,
        exc_value: BaseException | None,
        traceback: TracebackType | None,
        signal_number: int | None,
        stack_frame: FrameType | None
    ) -> str:
        """Return a string specifying the full origin of the error, as best as we can determine.

        This function has effectively two signatures. If the fatal error is caused by an exception,
        then the first 3 arguments will be used to match the signature of :func:`sys.excepthook`.
        If it's caused by a signal, then the last two will be used to match the signature of the
        handler in :func:`signal.signal`. This function should never be used outside this file, so
        we don't account for a mixture of arguments.

        :param exc_type: The type of the exception that caused the crash
        :param exc_value: The value of the exception itself
        :param traceback: The traceback object
        :param signal_number: The number of the signal that caused the crash
        :param stack_frame: The current stack frame object

        :type exc_type: Type[BaseException] | None
        :type exc_value: BaseException | None
        :type traceback: types.TracebackType | None
        :type signal_number: int | None
        :type stack_frame: types.FrameType | None
        """
        origin = 'CRASH ORIGIN:\n'

        if exc_type is not None and exc_value is not None and traceback is not None:
            # We want the frame where the exception actually occurred, so we have to descend the traceback
            # I don't know why we aren't given this traceback in the first place
            tb = traceback
            while tb.tb_next is not None:
                tb = tb.tb_next

            frame = tb.tb_frame

            origin += f'  Exception "{exc_value}"\n  of type {exc_type.__name__} in call to {frame.f_code.co_name}()\n'
            ↪  \
                f'  on line {frame.f_lineno} of {frame.f_code.co_filename}'

        elif signal_number is not None and stack_frame is not None:
            origin += f'  Signal "{signal.strsignal(signal_number)}" received in call to
            ↪  {stack_frame.f_code.co_name}()\n' \
                f'  on line {stack_frame.f_lineno} of {stack_frame.f_code.co_filename}'

        else:
            origin += '  UNKNOWN (not exception or signal)'

        origin += '\n\n'

        return origin


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

        string = 'Display settings:\n'

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

```

```
        return string


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

        try:
            matrix_wrapper = window._matrix_wrapper
            expression_history = window._expression_history
            exp_hist_index = window._expression_history_index
            plot = window._plot
            point_i = plot.point_i
            point_j = plot.point_j

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

        post_mortem = 'Matrix wrapper:\n'

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

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

            post_mortem += '\n'

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

        post_mortem += f'\nExpression history (index={exp_hist_index}):'
        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.2 `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
```

```
84
85
86  def update_lintrans() -> None:
87      """Update the lintrans binary executable, failing silently.
88
89      This function only makes sense if lintrans was installed, rather than being used as an executable.
90      We ask the :class:`~lintrans.global_settings.GlobalSettings` singleton where the executable is and,
91      if it exists, then we replace the old executable with the new one. This means that the next time
92      lintrans gets run, it will use the most recent version.
93
94      .. note::
95         This function doesn't care if the latest version on GitHub is actually newer than the current
96         version. Use :func:`new_version_exists` to check.
97      """
98      executable_path = GlobalSettings().get_executable_path()
99      if executable_path == '':
100         return
101
102     try:
103         html: str = urlopen('https://github.com/DoctorDalek1963/lintrans/releases/latest').read().decode()
104     except (UnicodeDecodeError, URLError):
105         return
106
107     match = re.search(
108         r'(?<=DoctorDalek1963/lintrans/releases/tag/v)\d+\.\d+\.\d+(?=;)',
109         html
110     )
111     if match is None:
112         return
113
114     latest_version = version.parse(match.group(0))
115
116     # We now know that the latest version is newer, and where the executable is,
117     # so we can begin the replacement process
118     url = 'https://github.com/DoctorDalek1963/lintrans/releases/download/'
119
120     if os.name == 'posix':
121         url += f'v{latest_version}/lintrans-Linux-{latest_version}'
122
123     elif os.name == 'nt':
124         url += f'v{latest_version}/lintrans-Windows-{latest_version}.exe'
125
126     else:
127         return
128
129     temp_file = GlobalSettings().get_update_download_filename()
130
131     # If the temp file already exists, then another instance of lintrans (probably
132     # in a background thread) is currently updating, so we don't want to interfere
133     if os.path.isfile(temp_file):
134         return
135
136     with open(temp_file, 'wb') as f:
137         try:
138             f.write(urlopen(url).read())
139         except URLError:
140             return
141
142     if os.name == 'posix':
143         os.rename(temp_file, executable_path)
144         subprocess.run(['chmod', '+x', executable_path])
145
146     elif os.name == 'nt':
147         # On Windows, we need to leave a process running in the background to automatically
148         # replace the exe file when lintrans stops running
149         script = '@echo off\n' \
150             ':loop\n\n' \
151             'timeout 5 >nul\n' \
152             'tasklist /fi "IMAGENAME eq lintrans.exe" /fo csv 2>nul | find /I "lintrans.exe" >nul\n' \
153             'if "%ERRORLEVEL%"=="0" goto :loop\n\n' \
154             f'del "{executable_path}"\n' \
155             f'rename "{temp_file}" lintrans.exe\n\n' \
156             '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.3 global_settings.py

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

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

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

from __future__ import annotations

import configparser
import os
import subprocess
import sys
from enum import Enum
from pathlib import Path

from singleton_decorator import singleton

_DEFAULT_CONFIG = '''
[General]
# Valid options are "auto", "prompt", or "never"
# An unknown option will default to "never"
Updates = prompt
'''[1:]


@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.
    """

    UpdateType = Enum('UpdateType', 'auto prompt never')

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

```
            self._directory = os.path.join(
                os.path.expanduser('~'),
                '.lintrans'
            )

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

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

        sub_directories = ['saves', 'crash_reports']

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

        self._executable_path = ''

        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

        self._settings_file = os.path.join(self._directory, 'settings.ini')
        config = configparser.ConfigParser()
        config.read(self._settings_file)

        try:
            self._general_settings = config['General']
        except KeyError:
            with open(self._settings_file, 'w', encoding='utf-8') as f:
                f.write(_DEFAULT_CONFIG)

            default_config = configparser.ConfigParser()
            default_config.read(self._settings_file)

            self._general_settings = default_config['General']

    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_executable_path(self) -> str:
        """Return the path to the binary executable, or an empty string if lintrans is not installed standalone."""
        return self._executable_path

    def get_update_type(self) -> UpdateType:
        """Return the update type defined in the settings file."""
        try:
            update_type = self._general_settings['Updates'].lower()
        except KeyError:
            return self.UpdateType.never

        # This is just to satisfy mypy and ensure that we return the Literal
        if update_type == 'auto':
            return self.UpdateType.auto
```

```
122
123             if update_type == 'prompt':
124                 return self.UpdateType.prompt
125
126             return self.UpdateType.never
127
128         def set_update_type(self, update_type: UpdateType) -> None:
129             """Set the update type in the settings file to the given type."""
130             self._general_settings['Updates'] = update_type.name
131
132             new_settings_file = _DEFAULT_CONFIG.replace(
133                 'Updates = prompt',
134                 f'Updates = {update_type.name}'
135             )
136
137             with open(self._settings_file, 'w', encoding='utf-8') as f:
138                 f.write(new_settings_file)
139
140         def get_settings_file(self) -> str:
141             """Return the full path of the settings file."""
142             return self._settings_file
143
144         def get_update_download_filename(self) -> str:
145             """Return a name for a temporary file next to the executable.
146
147             This method is used when downloading a new version of lintrans into a temporary file.
148             This is needed to allow :func:`os.rename` instead of :func:`shutil.move`. The first
149             requires the src and dest to be on the same partition, but also allows us to replace
150             the running executable.
151             """
152             return str(Path(self._executable_path).parent / 'lintrans-update-temp.dat')
153
154         def get_update_replace_bat_filename(self) -> str:
155             """Return the full path of the ``replace.bat`` file needed to update on Windows.
156
157             See :meth:`get_update_download_filename`.
158             """
159             return str(Path(self._executable_path).parent / 'replace.bat')
```

A.4 `__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.5 `__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__ = '0.4.0-alpha'
13
14  __all__ = ['crash_reporting', 'global_settings', 'gui', 'matrices', 'typing_', 'updating', '__version__']
```

A.6 gui/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>

"""This module provides the :class:`Session` class, which provides a way to save and load sessions."""

from __future__ import annotations

import os
import pathlib
import pickle
from collections import defaultdict
from typing import Any, DefaultDict, List, Tuple

import lintrans
from lintrans.gui.settings import DisplaySettings
from lintrans.matrices import MatrixWrapper


def _return_none() -> None:
    """Return None.

    This function only exists to make the defaultdict in :class:`Session` pickle-able.
    """
    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.7 gui/main_window.py

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

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

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

from __future__ import annotations

import os
import re
import sys
import webbrowser
from copy import deepcopy
from pathlib import Path
from pickle import UnpicklingError
from typing import List, NoReturn, Optional, Type

import numpy as np
from numpy import linalg
from numpy.linalg import LinAlgError
from PyQt5 import QtWidgets
from PyQt5.QtCore import QObject, Qt, QThread, pyqtSignal, pyqtSlot
from PyQt5.QtGui import QCloseEvent, QIcon, QKeyEvent, QKeySequence
from PyQt5.QtWidgets import (QAction, QApplication, QFileDialog, QHBoxLayout,
                             QMainWindow, QMenu, QMessageBox, QPushButton,
                             QShortcut, QSizePolicy, QSpacerItem,
                             QStyleFactory, QVBoxLayout)

import lintrans
from lintrans import updating
from lintrans.global_settings import GlobalSettings
from lintrans.matrices import MatrixWrapper
from lintrans.matrices.parse import validate_matrix_expression
from lintrans.matrices.utility import polar_coords, rotate_coord
```

```
37  from lintrans.typing_ import MatrixType, VectorType
38
39  from .dialogs import (AboutDialog, DefineAsExpressionDialog,
40                        DefineMatrixDialog, DefineNumericallyDialog,
41                        DefinePolygonDialog, DefineVisuallyDialog,
42                        DisplaySettingsDialog, FileSelectDialog, InfoPanelDialog,
43                        PromptUpdateDialog)
44  from .plots import MainViewportWidget
45  from .session import Session
46  from .settings import DisplaySettings
47  from .utility import qapp
48  from .validate import MatrixExpressionValidator
49
50
51  class _UpdateChecker(QObject):
52      """A simple class to act as a worker for a :class:`QThread`."""
53
54      signal_prompt_update: pyqtSignal = pyqtSignal(str)
55      """A signal that is emitted if a new version is found. The argument is the new version string."""
56
57      finished: pyqtSignal = pyqtSignal()
58      """A signal that is emitted when the worker has finished. Intended to be used for cleanup."""
59
60      def check_for_updates_and_emit(self) -> None:
61          """Check for updates, and emit :attr:`signal_prompt_update` if there's a new version.
62
63          This method exists to be run in a background thread to trigger a prompt if a new version is found.
64          """
65          update_type = GlobalSettings().get_update_type()
66
67          if update_type == GlobalSettings().UpdateType.never:
68              return
69
70          if update_type == GlobalSettings().UpdateType.auto:
71              updating.update_lintrans_in_background(check=True)
72              return
73
74          # If we get here, then update_type must be prompt,
75          # so we can check for updates and possibly prompt the user
76          new, version = updating.new_version_exists()
77          if new:
78              self.signal_prompt_update.emit(version)
79
80          self.finished.emit()
81
82
83  class LintransMainWindow(QMainWindow):
84      """This class provides a main window for the GUI using the Qt framework.
85
86      This class should not be used directly, instead call :func:`main` to create the GUI.
87      """
88
89      def __init__(self):
90          """Create the main window object, and create and arrange every widget in it.
91
92          This doesn't show the window, it just constructs it. Use :func:`main` to show the GUI.
93          """
94          super().__init__()
95
96          self._matrix_wrapper = MatrixWrapper()
97
98          self._expression_history: List[str] = []
99          self._expression_history_index: Optional[int] = None
100
101         self.setWindowTitle('lintrans')
102         self.setMinimumSize(800, 650)
103
104         path = Path(__file__).parent.absolute() / 'assets' / 'icon.jpg'
105         self.setWindowIcon(QIcon(str(path)))
106
107         self._animating: bool = False
108         self._animating_sequence: bool = False
109         self._reset_during_animation: bool = False
```

```
110
111             self._save_filename: Optional[str] = None
112
113             # Set up thread and worker to check for updates
114
115             self._thread_updates = QThread()
116             self._worker_updates = _UpdateChecker()
117             self._worker_updates.moveToThread(self._thread_updates)
118
119             self._thread_updates.started.connect(self._worker_updates.check_for_updates_and_emit)
120             self._worker_updates.signal_prompt_update.connect(self._prompt_update)
121             self._worker_updates.finished.connect(self._thread_updates.quit)
122             self._worker_updates.finished.connect(self._worker_updates.deleteLater)
123             self._thread_updates.finished.connect(self._thread_updates.deleteLater)
124
125             # === Create menubar
126
127             menubar = QtWidgets.QMenuBar(self)
128
129             menu_file = QMenu(menubar)
130             menu_file.setTitle('&File')
131
132             menu_help = QMenu(menubar)
133             menu_help.setTitle('&Help')
134
135             action_reset_session = QAction(self)
136             action_reset_session.setText('Reset session')
137             action_reset_session.triggered.connect(self._reset_session)
138
139             action_open = QAction(self)
140             action_open.setText('&Open')
141             action_open.setShortcut('Ctrl+O')
142             action_open.triggered.connect(self._ask_for_session_file)
143
144             action_save = QAction(self)
145             action_save.setText('&Save')
146             action_save.setShortcut('Ctrl+S')
147             action_save.triggered.connect(self._save_session)
148
149             action_save_as = QAction(self)
150             action_save_as.setText('Save as...')
151             action_save_as.setShortcut('Ctrl+Shift+S')
152             action_save_as.triggered.connect(self._save_session_as)
153
154             action_quit = QAction(self)
155             action_quit.setText('&Quit')
156             action_quit.triggered.connect(self.close)
157
158             # If this is an old release, use the docs for this release. Else, use the latest docs
159             # We use the latest because most use cases for non-stable releases will be in development and testing
160             docs_link = 'https://lintrans.readthedocs.io/en/'
161
162             if re.match(r'^\d+\.\d+\.\d+$', lintrans.__version__):
163                 docs_link += 'v' + lintrans.__version__
164             else:
165                 docs_link += 'latest'
166
167             action_tutorial = QAction(self)
168             action_tutorial.setText('&Tutorial')
169             action_tutorial.setShortcut('F1')
170             action_tutorial.triggered.connect(
171                 lambda: webbrowser.open_new_tab(docs_link + '/tutorial/index.html')
172             )
173
174             action_docs = QAction(self)
175             action_docs.setText('&Docs')
176             action_docs.triggered.connect(
177                 lambda: webbrowser.open_new_tab(docs_link + '/backend/lintrans.html')
178             )
179
180             menu_feedback = QMenu(menu_help)
181             menu_feedback.setTitle('Give feedback')
182
```

```
183             action_bug_report = QAction(self)
184             action_bug_report.setText('Report a bug')
185             action_bug_report.triggered.connect(
186                 lambda: webbrowser.open_new_tab('https://forms.gle/Q82cLTtgPLcV4xQD6')
187             )
188 
189             action_suggest_feature = QAction(self)
190             action_suggest_feature.setText('Suggest a new feature')
191             action_suggest_feature.triggered.connect(
192                 lambda: webbrowser.open_new_tab('https://forms.gle/mVWbHiMBw9Zq5Ze37')
193             )
194 
195             menu_feedback.addAction(action_bug_report)
196             menu_feedback.addAction(action_suggest_feature)
197 
198             action_about = QAction(self)
199             action_about.setText('&About')
200             action_about.triggered.connect(lambda: AboutDialog(self).open())
201 
202             menu_file.addAction(action_reset_session)
203             menu_file.addAction(action_open)
204             menu_file.addSeparator()
205             menu_file.addAction(action_save)
206             menu_file.addAction(action_save_as)
207             menu_file.addSeparator()
208             menu_file.addAction(action_quit)
209 
210             menu_help.addAction(action_tutorial)
211             menu_help.addAction(action_docs)
212             menu_help.addSeparator()
213             menu_help.addMenu(menu_feedback)
214             menu_help.addSeparator()
215             menu_help.addAction(action_about)
216 
217             menubar.addAction(menu_file.menuAction())
218             menubar.addAction(menu_help.menuAction())
219 
220             self.setMenuBar(menubar)
221 
222             # === Create widgets
223 
224             # Left layout: the plot and input box
225 
226             self._plot = MainViewportWidget(self, display_settings=DisplaySettings(), polygon_points=[])
227 
228             self._lineedit_expression_box = QtWidgets.QLineEdit(self)
229             self._lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
230             self._lineedit_expression_box.setValidator(MatrixExpressionValidator(self))
231             self._lineedit_expression_box.textChanged.connect(self._update_render_buttons)
232 
233             # Right layout: all the buttons
234 
235             # Misc buttons
236 
237             button_define_polygon = QPushButton(self)
238             button_define_polygon.setText('Define polygon')
239             button_define_polygon.clicked.connect(self._dialog_define_polygon)
240             button_define_polygon.setToolTip('Define a polygon to view its transformation<br><b>(Ctrl + P)</b>')
241             QShortcut(QKeySequence('Ctrl+P'), self).activated.connect(button_define_polygon.click)
242 
243             self._button_change_display_settings = QPushButton(self)
244             self._button_change_display_settings.setText('Change\ndisplay settings')
245             self._button_change_display_settings.clicked.connect(self._dialog_change_display_settings)
246             self._button_change_display_settings.setToolTip(
247                 "Change which things are rendered and how they're rendered<br><b>(Ctrl + D)</b>"
248             )
249             QShortcut(QKeySequence('Ctrl+D'), self).activated.connect(self._button_change_display_settings.click)
250 
251             button_reset_zoom = QPushButton(self)
252             button_reset_zoom.setText('Reset zoom')
253             button_reset_zoom.clicked.connect(self._reset_zoom)
254             button_reset_zoom.setToolTip('Reset the zoom level back to normal<br><b>(Ctrl + Shift + R)</b>')
255             QShortcut(QKeySequence('Ctrl+Shift+R'), self).activated.connect(button_reset_zoom.click)
```

```
256
257             # Define new matrix buttons and their groupbox
258
259             self._button_define_visually = QPushButton(self)
260             self._button_define_visually.setText('Visually')
261             self._button_define_visually.setToolTip('Drag the basis vectors<br><b>(Alt + 1)</b>')
262             self._button_define_visually.clicked.connect(lambda: self._dialog_define_matrix(DefineVisuallyDialog))
263             QShortcut(QKeySequence('Alt+1'), self).activated.connect(self._button_define_visually.click)
264
265             self._button_define_numerically = QPushButton(self)
266             self._button_define_numerically.setText('Numerically')
267             self._button_define_numerically.setToolTip('Define a matrix just with numbers<br><b>(Alt + 2)</b>')
268             self._button_define_numerically.clicked.connect(lambda: self._dialog_define_matrix(DefineNumericallyDialog))
269             QShortcut(QKeySequence('Alt+2'), self).activated.connect(self._button_define_numerically.click)
270
271             self._button_define_as_expression = QPushButton(self)
272             self._button_define_as_expression.setText('As an expression')
273             self._button_define_as_expression.setToolTip('Define a matrix in terms of other matrices<br><b>(Alt +
                ↪  3)</b>')
274             self._button_define_as_expression.clicked.connect(
275                 lambda: self._dialog_define_matrix(DefineAsExpressionDialog)
276             )
277             QShortcut(QKeySequence('Alt+3'), self).activated.connect(self._button_define_as_expression.click)
278
279             vlay_define_new_matrix = QVBoxLayout()
280             vlay_define_new_matrix.setSpacing(20)
281             vlay_define_new_matrix.addWidget(self._button_define_visually)
282             vlay_define_new_matrix.addWidget(self._button_define_numerically)
283             vlay_define_new_matrix.addWidget(self._button_define_as_expression)
284
285             groupbox_define_new_matrix = QtWidgets.QGroupBox('Define a new matrix', self)
286             groupbox_define_new_matrix.setLayout(vlay_define_new_matrix)
287
288             # Info panel button
289
290             self._button_info_panel = QPushButton(self)
291             self._button_info_panel.setText('Show defined matrices')
292             self._button_info_panel.clicked.connect(self._open_info_panel)
293             self._button_info_panel.setToolTip(
294                 'Open an info panel with all matrices that have been defined in this session<br><b>(Ctrl + M)</b>'
295             )
296             QShortcut(QKeySequence('Ctrl+M'), self).activated.connect(self._button_info_panel.click)
297
298             # Render buttons
299
300             button_reset = QPushButton(self)
301             button_reset.setText('Reset')
302             button_reset.clicked.connect(self._reset_transformation)
303             button_reset.setToolTip('Reset the visualized transformation back to the identity<br><b>(Ctrl + R)</b>')
304             QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(button_reset.click)
305
306             self._button_render = QPushButton(self)
307             self._button_render.setText('Render')
308             self._button_render.setEnabled(False)
309             self._button_render.clicked.connect(self._render_expression)
310             self._button_render.setToolTip('Render the expression<br><b>(Ctrl + Enter)</b>')
311             QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self._button_render.click)
312
313             self._button_animate = QPushButton(self)
314             self._button_animate.setText('Animate')
315             self._button_animate.setEnabled(False)
316             self._button_animate.clicked.connect(self._animate_expression)
317             self._button_animate.setToolTip('Animate the expression<br><b>(Ctrl + Shift + Enter)</b>')
318             QShortcut(QKeySequence('Ctrl+Shift+Return'), self).activated.connect(self._button_animate.click)
319
320             # === Arrange widgets
321
322             vlay_left = QVBoxLayout()
323             vlay_left.addWidget(self._plot)
324             vlay_left.addWidget(self._lineedit_expression_box)
325
326             vlay_misc_buttons = QVBoxLayout()
327             vlay_misc_buttons.setSpacing(20)
```

```
328             vlay_misc_buttons.addWidget(button_define_polygon)
329             vlay_misc_buttons.addWidget(self._button_change_display_settings)
330             vlay_misc_buttons.addWidget(button_reset_zoom)
331
332             vlay_info_buttons = QVBoxLayout()
333             vlay_info_buttons.setSpacing(20)
334             vlay_info_buttons.addWidget(self._button_info_panel)
335
336             vlay_render = QVBoxLayout()
337             vlay_render.setSpacing(20)
338             vlay_render.addWidget(button_reset)
339             vlay_render.addWidget(self._button_animate)
340             vlay_render.addWidget(self._button_render)
341
342             vlay_right = QVBoxLayout()
343             vlay_right.setSpacing(50)
344             vlay_right.addLayout(vlay_misc_buttons)
345             vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
346             vlay_right.addWidget(groupbox_define_new_matrix)
347             vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
348             vlay_right.addLayout(vlay_info_buttons)
349             vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
350             vlay_right.addLayout(vlay_render)
351
352             hlay_all = QHBoxLayout()
353             hlay_all.setSpacing(15)
354             hlay_all.addLayout(vlay_left)
355             hlay_all.addLayout(vlay_right)
356
357             central_widget = QtWidgets.QWidget()
358             central_widget.setLayout(hlay_all)
359             central_widget.setContentsMargins(10, 10, 10, 10)
360
361             self.setCentralWidget(central_widget)
362
363         def closeEvent(self, event: QCloseEvent) -> None:
364             """Handle a :class:`QCloseEvent` by confirming if the user wants to save, and cancelling animation."""
365             if self._save_filename is None or not self.isWindowModified():
366                 self._animating = False
367                 self._animating_sequence = False
368                 event.accept()
369                 return
370
371             dialog = QMessageBox(self)
372             dialog.setIcon(QMessageBox.Question)
373             dialog.setWindowTitle('Save changes?')
374             dialog.setText(f"If you don't save, then changes made to {self._save_filename} will be lost.")
375             dialog.setStandardButtons(QMessageBox.Save | QMessageBox.Discard | QMessageBox.Cancel)
376             dialog.setDefaultButton(QMessageBox.Save)
377
378             pressed_button = dialog.exec()
379
380             if pressed_button == QMessageBox.Save:
381                 self._save_session()
382
383             if pressed_button in (QMessageBox.Save, QMessageBox.Discard):
384                 self._animating = False
385                 self._animating_sequence = False
386                 event.accept()
387             else:
388                 event.ignore()
389
390         def keyPressEvent(self, event: QKeyEvent) -> None:
391             """Handle a :class:`QKeyEvent` by scrolling through expression history."""
392             key = event.key()
393
394             # Load previous expression
395             if key == Qt.Key_Up:
396                 if self._expression_history_index is None:
397                     if len(self._expression_history) == 0:
398                         event.ignore()
399                         return
400
```

```
                # If the index is none and we've got a history, set the index to -1
                self._expression_history_index = -1

            # If the index is in range of the list (the index is always negative), then decrement it
            elif self._expression_history_index > -len(self._expression_history):
                self._expression_history_index -= 1

            self._lineedit_expression_box.setText(self._expression_history[self._expression_history_index])

        # Load next expression
        elif key == Qt.Key_Down:
            if self._expression_history_index is None:
                event.ignore()
                return

            self._expression_history_index += 1

            # The index is always negative, so if we've reached 0, then we need to stop
            if self._expression_history_index == 0:
                self._expression_history_index = None
                self._lineedit_expression_box.setText('')
            else:
                self._lineedit_expression_box.setText(self._expression_history[self._expression_history_index])

        else:
            event.ignore()
            return

        event.accept()

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

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

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

            self._button_animate.setEnabled(valid)

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

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

    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:
```

```
474                 self._reset_during_animation = True
475
476             self._animating = False
477             self._animating_sequence = False
478
479             self._plot.plot_matrix(self._matrix_wrapper['I'])
480             self._plot.update()
481
482         @pyqtSlot()
483         def _render_expression(self) -> None:
484             """Render the transformation given by the expression in the input box."""
485             try:
486                 text = self._lineedit_expression_box.text()
487                 matrix = self._matrix_wrapper.evaluate_expression(text)
488
489             except LinAlgError:
490                 self._show_error_message('Singular matrix', 'Cannot take inverse of singular matrix.')
491                 return
492
493             self._extend_expression_history(text)
494
495             if self._is_matrix_too_big(matrix):
496                 return
497
498             self._plot.plot_matrix(matrix)
499             self._plot.update()
500
501         @pyqtSlot()
502         def _animate_expression(self) -> None:
503             """Animate from the current matrix to the matrix in the expression box."""
504             self._button_render.setEnabled(False)
505             self._button_animate.setEnabled(False)
506
507             matrix_start: MatrixType = np.array([
508                 [self._plot.point_i[0], self._plot.point_j[0]],
509                 [self._plot.point_i[1], self._plot.point_j[1]]
510             ])
511
512             text = self._lineedit_expression_box.text()
513
514             self._extend_expression_history(text)
515
516             # If there's commas in the expression, then we want to animate each part at a time
517             if ',' in text:
518                 current_matrix = matrix_start
519                 self._animating_sequence = True
520
521                 # For each expression in the list, right multiply it by the current matrix,
522                 # and animate from the current matrix to that new matrix
523                 for expr in text.split(',')[::-1]:
524                     if not self._animating_sequence:
525                         break
526
527                     try:
528                         new_matrix = self._matrix_wrapper.evaluate_expression(expr)
529
530                         if self._plot.display_settings.applicative_animation:
531                             new_matrix = new_matrix @ current_matrix
532                     except LinAlgError:
533                         self._show_error_message('Singular matrix', 'Cannot take inverse of singular matrix.')
534                         return
535
536                     self._animate_between_matrices(current_matrix, new_matrix)
537                     current_matrix = new_matrix
538
539                     # Here we just redraw and allow for other events to be handled while we pause
540                     self._plot.update()
541                     QApplication.processEvents()
542                     QThread.msleep(self._plot.display_settings.animation_pause_length)
543
544                 self._animating_sequence = False
545
546             # If there's no commas, then just animate directly from the start to the target
```

```
547             else:
548                 # Get the target matrix and its determinant
549                 try:
550                     matrix_target = self._matrix_wrapper.evaluate_expression(text)
551 
552                 except LinAlgError:
553                     self._show_error_message('Singular matrix', 'Cannot take inverse of singular matrix.')
554                     return
555 
556                 # The concept of applicative animation is explained in /gui/settings.py
557                 if self._plot.display_settings.applicative_animation:
558                     matrix_target = matrix_target @ matrix_start
559 
560                 # If we want a transitional animation and we're animating the same matrix, then restart the animation
561                 # We use this check rather than equality because of small floating point errors
562                 elif (abs(matrix_start - matrix_target) < 1e-12).all():
563                     matrix_start = self._matrix_wrapper['I']
564 
565                     # We pause here for 200 ms to make the animation look a bit nicer
566                     self._plot.plot_matrix(matrix_start)
567                     self._plot.update()
568                     QApplication.processEvents()
569                     QThread.msleep(200)
570 
571                 self._animate_between_matrices(matrix_start, matrix_target)
572 
573             self._update_render_buttons()
574 
575         def _get_animation_frame(self, start: MatrixType, target: MatrixType, proportion: float) -> MatrixType:
576             """Get the matrix to render for this frame of the animation.
577 
578             This method will smoothen the determinant if that setting in enabled and if the determinant is positive.
579             It also animates rotation-like matrices using a logarithmic spiral to rotate around and scale continuously.
580             Essentially, it just makes things look good when animating.
581 
582             :param MatrixType start: The starting matrix
583             :param MatrixType start: The target matrix
584             :param float proportion: How far we are through the loop
585             """
586             det_target = linalg.det(target)
587             det_start = linalg.det(start)
588 
589             # This is the matrix that we're applying to get from start to target
590             # We want to check if it's rotation-like
591             if linalg.det(start) == 0:
592                 matrix_application = None
593             else:
594                 matrix_application = target @ linalg.inv(start)
595 
596             # For a matrix to represent a rotation, it must have a positive determinant,
597             # its vectors must be perpendicular, the same length, and at right angles
598             # The checks for 'abs(value) < 1e-10' are to account for floating point error
599             if matrix_application is not None \
600                     and self._plot.display_settings.smoothen_determinant \
601                     and linalg.det(matrix_application) > 0 \
602                     and abs(np.dot(matrix_application.T[0], matrix_application.T[1])) < 1e-10 \
603                     and abs(np.hypot(*matrix_application.T[0]) - np.hypot(*matrix_application.T[1])) < 1e-10:
604                 rotation_vector: VectorType = matrix_application.T[0]  # Take the i column
605                 radius, angle = polar_coords(*rotation_vector)
606 
607                 # We want the angle to be in [-pi, pi), so we have to subtract 2pi from it if it's too big
608                 if angle > np.pi:
609                     angle -= 2 * np.pi
610 
611                 i: VectorType = start.T[0]
612                 j: VectorType = start.T[1]
613 
614                 # Scale the coords with a list comprehension
615                 # It's a bit janky, but rotate_coords() will always return a 2-tuple,
616                 # so new_i and new_j will always be lists of length 2
617                 scale = (radius - 1) * proportion + 1
618                 new_i = [scale * c for c in rotate_coord(i[0], i[1], angle * proportion)]
619                 new_j = [scale * c for c in rotate_coord(j[0], j[1], angle * proportion)]
```

```
620
621             return np.array(
622                 [
623                     [new_i[0], new_j[0]],
624                     [new_i[1], new_j[1]]
625                 ]
626             )
627
628         # matrix_a is the start matrix plus some part of the target, scaled by the proportion
629         # If we just used matrix_a, then things would animate, but the determinants would be weird
630         matrix_a = start + proportion * (target - start)
631
632         if not self._plot.display_settings.smoothen_determinant or det_start * det_target <= 0:
633             return matrix_a
634
635         # To fix the determinant problem, we get the determinant of matrix_a and use it to normalize
636         det_a = linalg.det(matrix_a)
637
638         # For a 2x2 matrix A and a scalar c, we know that det(cA) = c^2 det(A)
639         # We want B = cA such that det(B) = det(S), where S is the start matrix,
640         # so then we can scale it with the animation, so we get
641         # det(cA) = c^2 det(A) = det(S) => c = sqrt(abs(det(S) / det(A)))
642         # Then we scale A to get the determinant we want, and call that matrix_b
643         if det_a == 0:
644             c = 0
645         else:
646             c = np.sqrt(abs(det_start / det_a))
647
648         matrix_b = c * matrix_a
649         det_b = linalg.det(matrix_b)
650
651         # We want to return B, but we have to scale it over time to have the target determinant
652
653         # We want some C = dB such that det(C) is some target determinant T
654         # det(dB) = d^2 det(B) = T => d = sqrt(abs(T / det(B))
655
656         # We're also subtracting 1 and multiplying by the proportion and then adding one
657         # This just scales the determinant along with the animation
658
659         # That is all of course, if we can do that
660         # We'll crash if we try to do this with det(B) == 0
661         if det_b == 0:
662             return matrix_a
663
664         scalar = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
665         return scalar * matrix_b
666
667     def _animate_between_matrices(self, matrix_start: MatrixType, matrix_target: MatrixType) -> None:
668         """Animate from the start matrix to the target matrix."""
669         self._animating = True
670
671         # Making steps depend on animation_time ensures a smooth animation without
672         # massive overheads for small animation times
673         steps = self._plot.display_settings.animation_time // 10
674
675         for i in range(0, steps + 1):
676             if not self._animating:
677                 break
678
679             matrix_to_render = self._get_animation_frame(matrix_start, matrix_target, i / steps)
680
681             if self._is_matrix_too_big(matrix_to_render):
682                 self._animating = False
683                 self._animating_sequence = False
684                 return
685
686             self._plot.plot_matrix(matrix_to_render)
687
688             # We schedule the plot to be updated, tell the event loop to
689             # process events, and asynchronously sleep for 10ms
690             # This allows for other events to be processed while animating, like zooming in and out
691             self._plot.update()
692             QApplication.processEvents()
```

```
693                 QThread.msleep(self._plot.display_settings.animation_time // steps)
694
695             if not self._reset_during_animation:
696                 self._plot.plot_matrix(matrix_target)
697             else:
698                 self._plot.plot_matrix(self._matrix_wrapper['I'])
699
700             self._plot.update()
701
702             self._animating = False
703             self._reset_during_animation = False
704
705         @pyqtSlot()
706         def _open_info_panel(self) -> None:
707             """Open the info panel and register a callback to undefine matrices."""
708             dialog = InfoPanelDialog(self._matrix_wrapper, self)
709             dialog.open()
710             dialog.finished.connect(self._assign_matrix_wrapper)
711
712         @pyqtSlot(DefineMatrixDialog)
713         def _dialog_define_matrix(self, dialog_class: Type[DefineMatrixDialog]) -> None:
714             """Open a generic definition dialog to define a new matrix.
715
716             The class for the desired dialog is passed as an argument. We create an
717             instance of this class and the dialog is opened asynchronously and modally
718             (meaning it blocks interaction with the main window) with the proper method
719             connected to the :meth:`QDialog.accepted` signal.
720
721             .. note:: ``dialog_class`` must subclass
    ↪  :class:`~lintrans.gui.dialogs.define_new_matrix.DefineMatrixDialog`.
722
723             :param dialog_class: The dialog class to instantiate
724             :type dialog_class: Type[lintrans.gui.dialogs.define_new_matrix.DefineMatrixDialog]
725             """
726             # We create a dialog with a deepcopy of the current matrix_wrapper
727             # This avoids the dialog mutating this one
728             dialog: DefineMatrixDialog
729
730             if dialog_class == DefineVisuallyDialog:
731                 dialog = DefineVisuallyDialog(
732                     self,
733                     matrix_wrapper=deepcopy(self._matrix_wrapper),
734                     display_settings=self._plot.display_settings,
735                     polygon_points=self._plot.polygon_points
736                 )
737             else:
738                 dialog = dialog_class(self, matrix_wrapper=deepcopy(self._matrix_wrapper))
739
740             # .open() is asynchronous and doesn't spawn a new event loop, but the dialog is still modal (blocking)
741             dialog.open()
742
743             # So we have to use the accepted signal to call a method when the user accepts the dialog
744             dialog.accepted.connect(self._assign_matrix_wrapper)
745
746         @pyqtSlot()
747         def _assign_matrix_wrapper(self) -> None:
748             """Assign a new value to ``self._matrix_wrapper`` and give the expression box focus."""
749             self._matrix_wrapper = self.sender().matrix_wrapper
750             self._lineedit_expression_box.setFocus()
751             self._update_render_buttons()
752
753             self.setWindowModified(True)
754             self._update_window_title()
755
756         @pyqtSlot()
757         def _dialog_change_display_settings(self) -> None:
758             """Open the dialog to change the display settings."""
759             dialog = DisplaySettingsDialog(self, display_settings=self._plot.display_settings)
760             dialog.open()
761             dialog.accepted.connect(self._assign_display_settings)
762
763         @pyqtSlot()
764         def _assign_display_settings(self) -> None:
```

```
765             """Assign a new value to ``self._plot.display_settings`` and give the expression box focus."""
766             self._plot.display_settings = self.sender().display_settings
767             self._plot.update()
768             self._lineedit_expression_box.setFocus()
769             self._update_render_buttons()
770 
771         @pyqtSlot()
772         def _dialog_define_polygon(self) -> None:
773             """Open the dialog to define a polygon."""
774             dialog = DefinePolygonDialog(self, polygon_points=self._plot.polygon_points)
775             dialog.open()
776             dialog.accepted.connect(self._assign_polygon_points)
777 
778         @pyqtSlot()
779         def _assign_polygon_points(self) -> None:
780             """Assign a new value to ``self._plot.polygon_points`` and give the expression box focus."""
781             self._plot.polygon_points = self.sender().polygon_points
782             self._plot.update()
783             self._lineedit_expression_box.setFocus()
784             self._update_render_buttons()
785 
786             self.setWindowModified(True)
787             self._update_window_title()
788 
789         def _show_error_message(self, title: str, text: str, info: str | None = None, *, warning: bool = False) -> None:
790             """Show an error message in a dialog box.
791 
792             :param str title: The window title of the dialog box
793             :param str text: The simple error message
794             :param info: The more informative error message
795             :type info: Optional[str]
796             """
797             dialog = QMessageBox(self)
798             dialog.setWindowTitle(title)
799             dialog.setText(text)
800 
801             if warning:
802                 dialog.setIcon(QMessageBox.Warning)
803             else:
804                 dialog.setIcon(QMessageBox.Critical)
805 
806             if info is not None:
807                 dialog.setInformativeText(info)
808 
809             dialog.open()
810 
811             # This is `finished` rather than `accepted` because we want to update the buttons no matter what
812             dialog.finished.connect(self._update_render_buttons)
813 
814         def _is_matrix_too_big(self, matrix: MatrixType) -> bool:
815             """Check if the given matrix will actually fit on the grid.
816 
817             We're checking against a 1000x1000 grid here, which is far less than the actual space we have available.
818             But even when fully zoomed out 1080p monitor, the grid is only roughly 170x90, so 1000x1000 is plenty.
819 
820             :param MatrixType matrix: The matrix to check
821             :returns bool: Whether the matrix is too big to fit on the canvas
822             """
823             for x, y in matrix.T:
824                 if not (-1000 <= x <= 1000 and -1000 <= y <= 1000):
825                     self._show_error_message(
826                         'Matrix too big',
827                         "This matrix doesn't fit on the grid.",
828                         'This grid is only 1000x1000, and this matrix\n'
829                         f'[{int(matrix[0][0])} {int(matrix[0][1])}; {int(matrix[1][0])} {int(matrix[1][1])}]\n'
830                         " doesn't fit."
831                     )
832                     return True
833 
834             return False
835 
836         def _update_window_title(self) -> None:
837             """Update the window title to reflect whether the session has changed since it was last saved."""
```

```
838             title = 'lintrans'
839
840             if self._save_filename:
841                 title = os.path.split(self._save_filename)[-1] + '[*] - ' + title
842
843             self.setWindowTitle(title)
844
845         def _reset_session(self) -> None:
846             """Ask the user if they want to reset the current session.
847
848             Resetting the session means setting the matrix wrapper to a new instance, and rendering I.
849             """
850             dialog = QMessageBox(self)
851             dialog.setIcon(QMessageBox.Question)
852             dialog.setWindowTitle('Reset the session?')
853             dialog.setText('Are you sure you want to reset the current session?')
854             dialog.setStandardButtons(QMessageBox.Yes | QMessageBox.No)
855             dialog.setDefaultButton(QMessageBox.No)
856
857             if dialog.exec() == QMessageBox.Yes:
858                 self._matrix_wrapper = MatrixWrapper()
859                 self._plot.polygon_points = []
860
861                 self._reset_transformation()
862                 self._expression_history = []
863                 self._expression_history_index = None
864                 self._lineedit_expression_box.setText('')
865                 self._lineedit_expression_box.setFocus()
866                 self._update_render_buttons()
867
868                 self._save_filename = None
869                 self.setWindowModified(False)
870                 self._update_window_title()
871
872         def open_session_file(self, filename: str) -> None:
873             """Open the given session file.
874
875             If the selected file is not a valid lintrans session file, we just show an error message,
876             but if it's valid, we load it and set it as the default filename for saving.
877             """
878             try:
879                 session, version, extra_attrs = Session.load_from_file(filename)
880
881             # load_from_file() can raise errors if the contents is not a valid pickled Python object,
882             # or if the pickled Python object is of the wrong type
883             except (AttributeError, EOFError, FileNotFoundError, ValueError, UnpicklingError):
884                 self._show_error_message(
885                     'Invalid file contents',
886                     'This is not a valid lintrans session file.',
887                     'Not all .lt files are lintrans session files. This file was probably created by an unrelated '
888                     'program.'
889                 )
890                 return
891
892             missing_parts = False
893
894             if session.matrix_wrapper is not None:
895                 self._matrix_wrapper = session.matrix_wrapper
896             else:
897                 self._matrix_wrapper = MatrixWrapper()  # type: ignore[unreachable]
898                 missing_parts = True
899
900             if session.polygon_points is not None:
901                 self._plot.polygon_points = session.polygon_points
902             else:
903                 self._plot.polygon_points = []  # type: ignore[unreachable]
904                 missing_parts = True
905
906             if session.display_settings is not None:
907                 self._plot.display_settings = session.display_settings
908             else:
909                 self._plot.display_settings = DisplaySettings()  # type: ignore[unreachable]
910                 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')

            if dialog.exec():
                filename = dialog.selectedFiles()[0]
                self._save_filename = filename
                self._save_session()

        @pyqtSlot(str)
        def _prompt_update(self, version: str) -> None:
            """Open a modal dialog to prompt the user to update lintrans."""
            dialog = PromptUpdateDialog(self, new_version=version)
            dialog.open()

        def check_for_updates_and_prompt(self) -> None:
            """Update lintrans depending on the user's choice of update type.

            If they chose 'prompt', then this method will open a prompt dialog (after checking
            if a new version actually exists). See :meth:`_prompt_update`.
            """
            self._thread_updates.start()


    def main(filename: Optional[str]) -> NoReturn:
        """Run the GUI by creating and showing an instance of :class:`LintransMainWindow`.

        :param Optional[str] filename: A session file to optionally open at startup
        """
        app = QApplication([])
        app.setApplicationName('lintrans')
        app.setApplicationVersion(lintrans.__version__)

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

        window = LintransMainWindow()
        window.show()
        window.check_for_updates_and_prompt()

        if filename:
            window.open_session_file(filename)

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

A.8 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.9 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.10 gui/utility.py

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

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

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

from PyQt5.QtCore import QCoreApplication

```

```

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

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

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

    return instance
```

A.11 gui/settings.py

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

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

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

from __future__ import annotations

from dataclasses import dataclass


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

    # === Basic stuff

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

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

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

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

    label_basis_vectors: bool = False
    """This controls whether we want to label the `i` and `j` basis vectors."""

    # === Animations

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

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

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

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

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

```
56
57          animation_pause_length: int = 400
58          """This is the number of milliseconds that we wait between animations when using comma syntax."""
59
60          # === Matrix info
61
62          draw_determinant_parallelogram: bool = False
63          """This controls whether or not we should shade the parallelogram representing the determinant of the matrix."""
64
65          show_determinant_value: bool = True
66          """This controls whether we should write the text value of the determinant inside the parallelogram.
67
68          The text only gets draw if :attr:`draw_determinant_parallelogram` is also True.
69          """
70
71          draw_eigenvectors: bool = False
72          """This controls whether we should draw the eigenvectors of the transformation."""
73
74          draw_eigenlines: bool = False
75          """This controls whether we should draw the eigenlines of the transformation."""
76
77          # === Polygon
78
79          draw_untransformed_polygon: bool = True
80          """This controls whether we should draw the untransformed version of the user-defined polygon."""
81
82          draw_transformed_polygon: bool = True
83          """This controls whether we should draw the transformed version of the user-defined polygon."""
84
85          # === Input/output vectors
86
87          draw_input_vector: bool = True
88          """This controls whether we should draw the input vector in the main viewport."""
89
90          draw_output_vector: bool = True
91          """This controls whether we should draw the output vector in the main viewport."""
```

A.12 gui/plots/widgets.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7   """This module provides the actual widgets that can be used to visualize transformations in the GUI."""
8
9   from __future__ import annotations
10
11  import operator
12  from abc import abstractmethod
13  from math import dist
14  from typing import List, Optional, Tuple
15
16  from PyQt5.QtCore import QPointF, Qt, pyqtSlot
17  from PyQt5.QtGui import (QBrush, QColor, QMouseEvent, QPainter, QPaintEvent,
18                           QPen, QPolygonF)
19
20  from lintrans.gui.settings import DisplaySettings
21  from lintrans.typing_ import MatrixType
22
23  from .classes import InteractivePlot, VisualizeTransformationPlot
24
25
26  class VisualizeTransformationWidget(VisualizeTransformationPlot):
27      """This widget is used in the main window to visualize transformations.
28
29      It handles all the rendering itself, and the only method that the user needs to care about
30      is :meth:`plot_matrix`, which allows you to visualize the given matrix transformation.
31      """
```

```
32
33          def __init__(self, *args, display_settings: DisplaySettings, polygon_points: List[Tuple[float, float]],
            ↪  **kwargs):
34              """Create the widget and assign its display settings, passing ``*args`` and ``**kwargs`` to super."""
35              super().__init__(*args, **kwargs)
36
37              self.display_settings = display_settings
38              self.polygon_points = polygon_points
39
40          def plot_matrix(self, matrix: MatrixType) -> None:
41              """Plot the given matrix on the grid by setting the basis vectors.
42
43              .. warning:: This method does not call :meth:`QWidget.update()`. This must be done by the caller.
44
45              :param MatrixType matrix: The matrix to plot
46              """
47              self.point_i = (matrix[0][0], matrix[1][0])
48              self.point_j = (matrix[0][1], matrix[1][1])
49
50          def _draw_scene(self, painter: QPainter) -> None:
51              """Draw the default scene of the transformation.
52
53              This method exists to make it easier to split the main viewport from visual definitions while
54              not using multiple :class:`QPainter` objects from a single :meth:`paintEvent` call in a subclass.
55              """
56              painter.setRenderHint(QPainter.Antialiasing)
57              painter.setBrush(Qt.NoBrush)
58
59              self._draw_background(painter, self.display_settings.draw_background_grid)
60
61              if self.display_settings.draw_eigenlines:
62                  self._draw_eigenlines(painter)
63
64              if self.display_settings.draw_eigenvectors:
65                  self._draw_eigenvectors(painter)
66
67              if self.display_settings.draw_determinant_parallelogram:
68                  self._draw_determinant_parallelogram(painter)
69
70                  if self.display_settings.show_determinant_value:
71                      self._draw_determinant_text(painter)
72
73              if self.display_settings.draw_transformed_grid:
74                  self._draw_transformed_grid(painter)
75
76              if self.display_settings.draw_basis_vectors:
77                  self._draw_basis_vectors(painter)
78
79                  if self.display_settings.label_basis_vectors:
80                      self._draw_basis_vector_labels(painter)
81
82              if self.display_settings.draw_untransformed_polygon:
83                  self._draw_untransformed_polygon(painter)
84
85              if self.display_settings.draw_transformed_polygon:
86                  self._draw_transformed_polygon(painter)
87
88          @abstractmethod
89          def paintEvent(self, event: QPaintEvent) -> None:
90              """Paint the scene of the transformation."""
91
92
93      class MainViewportWidget(VisualizeTransformationWidget, InteractivePlot):
94          """This is the widget for the main viewport.
95
96          It extends :class:`VisualizeTransformationWidget` with input and output vectors.
97          """
98
99          _COLOUR_OUTPUT_VECTOR = QColor('#f7c216')
100
101         def __init__(self, *args, **kwargs):
102             """Create the main viewport widget with its input point."""
103             super().__init__(*args,  **kwargs)
```

```
104
105             self.point_input_vector: Tuple[float, float] = (1, 1)
106             self._dragging_vector: bool = False
107
108         def _draw_input_vector(self, painter: QPainter) -> None:
109             """Draw the input vector."""
110             pen = QPen(QColor('#000000'), self._WIDTH_VECTOR_LINE)
111             painter.setPen(pen)
112
113             x, y = self.canvas_coords(*self.point_input_vector)
114             painter.drawLine(*self._canvas_origin, x, y)
115
116             painter.setBrush(self._BRUSH_SOLID_WHITE)
117
118             painter.setPen(Qt.NoPen)
119             painter.drawPie(
120                 x - self._CURSOR_EPSILON,
121                 y - self._CURSOR_EPSILON,
122                 2 * self._CURSOR_EPSILON,
123                 2 * self._CURSOR_EPSILON,
124                 0,
125                 16 * 360
126             )
127
128             painter.setPen(pen)
129             painter.drawArc(
130                 x - self._CURSOR_EPSILON,
131                 y - self._CURSOR_EPSILON,
132                 2 * self._CURSOR_EPSILON,
133                 2 * self._CURSOR_EPSILON,
134                 0,
135                 16 * 360
136             )
137
138         def _draw_output_vector(self, painter: QPainter) -> None:
139             """Draw the output vector."""
140             painter.setPen(QPen(self._COLOUR_OUTPUT_VECTOR, self._WIDTH_VECTOR_LINE))
141             painter.setBrush(QBrush(self._COLOUR_OUTPUT_VECTOR, Qt.SolidPattern))
142
143             x, y = self.canvas_coords(*(self._matrix @ self.point_input_vector))
144
145             painter.drawLine(*self._canvas_origin, x, y)
146             painter.drawPie(
147                 x - self._CURSOR_EPSILON,
148                 y - self._CURSOR_EPSILON,
149                 2 * self._CURSOR_EPSILON,
150                 2 * self._CURSOR_EPSILON,
151                 0,
152                 16 * 360
153             )
154
155         def paintEvent(self, event: QPaintEvent) -> None:
156             """Paint the scene by just calling :meth:`_draw_scene` and drawing the I/O vectors."""
157             painter = QPainter()
158             painter.begin(self)
159
160             self._draw_scene(painter)
161
162             if self.display_settings.draw_output_vector:
163                 self._draw_output_vector(painter)
164
165             if self.display_settings.draw_input_vector:
166                 self._draw_input_vector(painter)
167
168             painter.end()
169             event.accept()
170
171         def mousePressEvent(self, event: QMouseEvent) -> None:
172             """Check if the user has clicked on the input vector."""
173             cursor_pos = (event.x(), event.y())
174
175             if event.button() != Qt.LeftButton:
176                 event.ignore()
```

```
177                 return
178
179             if self._is_within_epsilon(cursor_pos, self.point_input_vector):
180                 self._dragging_vector = True
181
182             event.accept()
183
184         def mouseReleaseEvent(self, event: QMouseEvent) -> None:
185             """Stop dragging the input vector."""
186             if event.button() == Qt.LeftButton:
187                 self._dragging_vector = False
188                 event.accept()
189             else:
190                 event.ignore()
191
192         def mouseMoveEvent(self, event: QMouseEvent) -> None:
193             """Drag the input vector if the user has clicked on it."""
194             if not self._dragging_vector:
195                 event.ignore()
196                 return
197
198             x, y = self._round_to_int_coord(self._grid_coords(event.x(), event.y()))
199             self.point_input_vector = (x, y)
200
201             self.update()
202             event.accept()
203
204
205     class DefineMatrixVisuallyWidget(VisualizeTransformationWidget, InteractivePlot):
206         """This widget allows the user to visually define a matrix.
207
208         This is just the widget itself. If you want the dialog, use
209         :class:`~lintrans.gui.dialogs.define_new_matrix.DefineVisuallyDialog`.
210         """
211
212         def __init__(self, *args, display_settings: DisplaySettings, polygon_points: List[Tuple[float, float]],
        ↪  **kwargs):
213             """Create the widget and enable mouse tracking. ``*args`` and ``**kwargs`` are passed to ``super()``."""
214             super().__init__(*args, display_settings=display_settings, polygon_points=polygon_points, **kwargs)
215
216             self._dragged_point: Tuple[float, float] | None = None
217
218         def paintEvent(self, event: QPaintEvent) -> None:
219             """Paint the scene by just calling :meth:`_draw_scene`."""
220             painter = QPainter()
221             painter.begin(self)
222
223             self._draw_scene(painter)
224
225             painter.end()
226             event.accept()
227
228         def mousePressEvent(self, event: QMouseEvent) -> None:
229             """Set the dragged point if the cursor is within :attr:`_CURSOR_EPSILON`."""
230             cursor_pos = (event.x(), event.y())
231
232             if event.button() != Qt.LeftButton:
233                 event.ignore()
234                 return
235
236             for point in (self.point_i, self.point_j):
237                 if self._is_within_epsilon(cursor_pos, point):
238                     self._dragged_point = point[0], point[1]
239
240             event.accept()
241
242         def mouseReleaseEvent(self, event: QMouseEvent) -> None:
243             """Handle the mouse click being released by unsetting the dragged point."""
244             if event.button() == Qt.LeftButton:
245                 self._dragged_point = None
246                 event.accept()
247             else:
248                 event.ignore()
```

```python

    def mouseMoveEvent(self, event: QMouseEvent) -> None:
        """Handle the mouse moving on the canvas."""
        if self._dragged_point is None:
            event.ignore()
            return

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

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

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

        self._dragged_point = x, y

        self.update()
        event.accept()


class DefinePolygonWidget(InteractivePlot):
    """This widget allows the user to define a polygon by clicking and dragging points on the canvas."""

    def __init__(self, *args, polygon_points: List[Tuple[float, float]], **kwargs):
        """Create the widget with a list of points and a dragged point index."""
        super().__init__(*args, **kwargs)

        self._dragged_point_index: Optional[int] = None
        self.points = polygon_points.copy()

    @pyqtSlot()
    def reset_polygon(self) -> None:
        """Reset the polygon and update the widget."""
        self.points = []
        self.update()

    def mousePressEvent(self, event: QMouseEvent) -> None:
        """Handle the mouse being clicked by adding a point or setting the dragged point index to an existing
        ↪  point."""
        if event.button() not in (Qt.LeftButton, Qt.RightButton):
            event.ignore()
            return

        canvas_pos = (event.x(), event.y())
        grid_pos = self._grid_coords(*canvas_pos)

        if event.button() == Qt.LeftButton:
            for i, point in enumerate(self.points):
                if self._is_within_epsilon(canvas_pos, point):
                    self._dragged_point_index = i
                    event.accept()
                    return

            new_point = self._round_to_int_coord(grid_pos)

            if len(self.points) < 2:
                self.points.append(new_point)
                self._dragged_point_index = -1
            else:
                # FIXME: This algorithm doesn't work very well when the new point is far away
                # from the existing polygon; it just picks the longest side

                # Get a list of line segments and a list of their lengths
                line_segments = list(zip(self.points, self.points[1:])) + [(self.points[-1], self.points[0])]
                segment_lengths = map(lambda t: dist(*t), line_segments)

                # Get the distance from each point in the polygon to the new point
                distances_to_point = [dist(p, new_point) for p in self.points]

                # For each pair of list-adjacent points, zip their distances to
                # the new point into a tuple, and add them together
                # This gives us the lengths of the catheti of the triangles that
```

```
                # connect the new point to each pair of adjacent points
                dist_to_point_pairs = list(zip(distances_to_point, distances_to_point[1:])) + \
                    [(distances_to_point[-1], distances_to_point[0])]

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

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

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

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

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

        self.update()
        event.accept()

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

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

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

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

        self.update()

        event.accept()

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

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

        painter.setBrush(self._BRUSH_SOLID_WHITE)

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

            painter.setPen(Qt.NoPen)
            painter.drawPie(
                x - self._CURSOR_EPSILON,
                y - self._CURSOR_EPSILON,
```

```
                2 * self._CURSOR_EPSILON,
                2 * self._CURSOR_EPSILON,
                0,
                16 * 360
            )

            painter.setPen(self._PEN_POLYGON)
            painter.drawArc(
                x - self._CURSOR_EPSILON,
                y - self._CURSOR_EPSILON,
                2 * self._CURSOR_EPSILON,
                2 * self._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.13 gui/plots/classes.py

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

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

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

from __future__ import annotations

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

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

from lintrans.typing_ import MatrixType, VectorType


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

    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.
    """

    DEFAULT_GRID_SPACING: int = 85
    """This is the starting spacing between grid lines (in pixels)."""

```

```
36          _MINIMUM_GRID_SPACING: int = 5
37          """This is the minimum spacing between grid lines (in pixels)."""
38
39          _COLOUR_BACKGROUND_GRID: QColor = QColor('#808080')
40          """This is the colour of the background grid lines."""
41
42          _COLOUR_BACKGROUND_AXES: QColor = QColor('#000000')
43          """This is the colour of the background axes."""
44
45          _WIDTH_BACKGROUND_GRID: float = 0.3
46          """This is the width of the background grid lines, as a multiple of the :class:`QPainter` line width."""
47
48          _PEN_POLYGON: QPen = QPen(QColor('#000000'), 1.5)
49          """This is the pen used to draw the normal polygon."""
50
51          _BRUSH_SOLID_WHITE: QBrush = QBrush(QColor('#FFFFFF'), Qt.SolidPattern)
52          """This brush is just solid white. Used to draw the insides of circles."""
53
54          def __init__(self, *args, **kwargs):
55              """Create the widget and setup backend stuff for rendering.
56
57              .. note:: ``*args`` and ``**kwargs`` are passed the superclass constructor (:class:`QWidget`).
58              """
59              super().__init__(*args, **kwargs)
60
61              self.setAutoFillBackground(True)
62
63              # Set the background to white
64              palette = self.palette()
65              palette.setColor(self.backgroundRole(), Qt.white)
66              self.setPalette(palette)
67
68              self.grid_spacing = self.DEFAULT_GRID_SPACING
69
70          @property
71          def _canvas_origin(self) -> Tuple[int, int]:
72              """Return the canvas coords of the grid origin.
73
74              The return value is intended to be unpacked and passed to a :meth:`QPainter.drawLine:iiii` call.
75
76              See :meth:`canvas_coords`.
77
78              :returns: The canvas coordinates of the grid origin
79              :rtype: Tuple[int, int]
80              """
81              return self.width() // 2, self.height() // 2
82
83          def _canvas_x(self, x: float) -> int:
84              """Convert an x coordinate from grid coords to canvas coords."""
85              return int(self._canvas_origin[0] + x * self.grid_spacing)
86
87          def _canvas_y(self, y: float) -> int:
88              """Convert a y coordinate from grid coords to canvas coords."""
89              return int(self._canvas_origin[1] - y * self.grid_spacing)
90
91          def canvas_coords(self, x: float, y: float) -> Tuple[int, int]:
92              """Convert a coordinate from grid coords to canvas coords.
93
94              This method is intended to be used like
95
96              .. code::
97
98                 painter.drawLine(*self.canvas_coords(x1, y1), *self.canvas_coords(x2, y2))
99
100             or like
101
102             .. code::
103
104                painter.drawLine(*self._canvas_origin, *self.canvas_coords(x, y))
105
106             See :attr:`_canvas_origin`.
107
108             :param float x: The x component of the grid coordinate
```

```
109             :param float y: The y component of the grid coordinate
110             :returns: The resultant canvas coordinates
111             :rtype: Tuple[int, int]
112             """
113             return self._canvas_x(x), self._canvas_y(y)
114 
115         def _grid_corner(self) -> Tuple[float, float]:
116             """Return the grid coords of the top right corner."""
117             return self.width() / (2 * self.grid_spacing), self.height() / (2 * self.grid_spacing)
118 
119         def _grid_coords(self, x: int, y: int) -> Tuple[float, float]:
120             """Convert a coordinate from canvas coords to grid coords.
121 
122             :param int x: The x component of the canvas coordinate
123             :param int y: The y component of the canvas coordinate
124             :returns: The resultant grid coordinates
125             :rtype: Tuple[float, float]
126             """
127             # We get the maximum grid coords and convert them into canvas coords
128             return (x - self._canvas_origin[0]) / self.grid_spacing, (-y + self._canvas_origin[1]) / self.grid_spacing
129 
130         @abstractmethod
131         def paintEvent(self, event: QPaintEvent) -> None:
132             """Handle a :class:`QPaintEvent`.
133 
134             .. note:: This method is abstract and must be overridden by all subclasses.
135             """
136 
137         def _draw_background(self, painter: QPainter, draw_grid: bool) -> None:
138             """Draw the background grid.
139 
140             .. note:: This method is just a utility method for subclasses to use to render the background grid.
141 
142             :param QPainter painter: The painter to draw the background with
143             :param bool draw_grid: Whether to draw the grid lines
144             """
145             if draw_grid:
146                 painter.setPen(QPen(self._COLOUR_BACKGROUND_GRID, self._WIDTH_BACKGROUND_GRID))
147 
148                 # Draw equally spaced vertical lines, starting in the middle and going out
149                 # We loop up to half of the width. This is because we draw a line on each side in each iteration
150                 for x in range(self.width() // 2 + self.grid_spacing, self.width(), self.grid_spacing):
151                     painter.drawLine(x, 0, x, self.height())
152                     painter.drawLine(self.width() - x, 0, self.width() - x, self.height())
153 
154                 # Same with the horizontal lines
155                 for y in range(self.height() // 2 + self.grid_spacing, self.height(), self.grid_spacing):
156                     painter.drawLine(0, y, self.width(), y)
157                     painter.drawLine(0, self.height() - y, self.width(), self.height() - y)
158 
159             # Now draw the axes
160             painter.setPen(QPen(self._COLOUR_BACKGROUND_AXES, self._WIDTH_BACKGROUND_GRID))
161             painter.drawLine(self.width() // 2, 0, self.width() // 2, self.height())
162             painter.drawLine(0, self.height() // 2, self.width(), self.height() // 2)
163 
164         def wheelEvent(self, event: QWheelEvent) -> None:
165             """Handle a :class:`QWheelEvent` by zooming in or our of the grid."""
166             # angleDelta() returns a number of units equal to 8 times the number of degrees rotated
167             degrees = event.angleDelta() / 8
168 
169             if degrees is not None:
170                 new_spacing = max(1, self.grid_spacing + degrees.y())
171 
172                 if new_spacing >= self._MINIMUM_GRID_SPACING:
173                     self.grid_spacing = new_spacing
174 
175             event.accept()
176             self.update()
177 
178 
179     class InteractivePlot(BackgroundPlot):
180         """This class represents an interactive plot, which allows the user to click and/or drag point(s).
181 
```

```
182         It declares the Qt methods needed for mouse cursor interaction to be abstract,
183         requiring all subclasses to implement these.
184         """
185
186         _CURSOR_EPSILON: int = 5
187         """This is the distance (in pixels) that the cursor needs to be from the point to drag it."""
188
189         _SNAP_DIST = 0.1
190         """This is the distance (in grid coords) that the cursor needs to be from an integer point to snap to it."""
191
192         def _round_to_int_coord(self, point: Tuple[float, float]) -> Tuple[float, float]:
193             """Take a coordinate in grid coords and round it to an integer coordinate if it's within :attr:`_SNAP_DIST`.
194
195             If the point is not close enough, we just return the original point.
196             """
197             x, y = point
198
199             possible_snaps: List[Tuple[int, int]] = [
200                 (floor(x), floor(y)),
201                 (floor(x), ceil(y)),
202                 (ceil(x), floor(y)),
203                 (ceil(x), ceil(y))
204             ]
205
206             snap_distances: List[Tuple[float, Tuple[int, int]]] = [
207                 (dist((x, y), coord), coord)
208                 for coord in possible_snaps
209             ]
210
211             for snap_dist, coord in snap_distances:
212                 if snap_dist < self._SNAP_DIST:
213                     x, y = coord
214
215             return x, y
216
217         def _is_within_epsilon(self, cursor_pos: Tuple[float, float], point: Tuple[float, float]) -> bool:
218             """Check if the cursor position (in canvas coords) is within range of the given point."""
219             mx, my = cursor_pos
220             px, py = self.canvas_coords(*point)
221             return (abs(px - mx) <= self._CURSOR_EPSILON and abs(py - my) <= self._CURSOR_EPSILON)
222
223         @abstractmethod
224         def mousePressEvent(self, event: QMouseEvent) -> None:
225             """Handle the mouse being pressed."""
226
227         @abstractmethod
228         def mouseReleaseEvent(self, event: QMouseEvent) -> None:
229             """Handle the mouse being released."""
230
231         @abstractmethod
232         def mouseMoveEvent(self, event: QMouseEvent) -> None:
233             """Handle the mouse moving on the widget."""
234
235
236     class VectorGridPlot(BackgroundPlot):
237         """This class represents a background plot, with vectors and their grid drawn on top. It provides utility
        ↪  methods.
238
239         .. note::
240             This is a simple superclass for vectors and is not for visualizing transformations.
241             See :class:`VisualizeTransformationPlot`.
242
243         This class should be subclassed to be used for visualization and matrix definition widgets.
244         All useful behaviour should be implemented by any subclass.
245
246         .. warning:: This class should never be directly instantiated, only subclassed.
247         """
248
249         _COLOUR_I = QColor('#0808d8')
250         """This is the colour of the `i` basis vector and associated transformed grid lines."""
251
252         _COLOUR_J = QColor('#e90000')
253         """This is the colour of the `j` basis vector and associated transformed grid lines."""
```

```
254
255         _COLOUR_TEXT = QColor('#000000')
256         """This is the colour of the text."""
257
258         _WIDTH_VECTOR_LINE = 1.8
259         """This is the width of the transformed basis vector lines, as a multiple of the :class:`QPainter` line
            ↪  width."""
260
261         _WIDTH_TRANSFORMED_GRID = 0.8
262         """This is the width of the transformed grid lines, as a multiple of the :class:`QPainter` line width."""
263
264         _ARROWHEAD_LENGTH = 0.15
265         """This is the minimum length (in grid coord size) of the arrowhead parts."""
266
267         _MAX_PARALLEL_LINES = 150
268         """This is the maximum number of parallel transformed grid lines that will be drawn.
269
270         The user can zoom out further, but we will stop drawing grid lines beyond this number.
271         """
272
273         def __init__(self, *args, **kwargs):
274             """Create the widget with ``point_i`` and ``point_j`` attributes.
275
276             .. note:: ``*args`` and ``**kwargs`` are passed to the superclass constructor (:class:`BackgroundPlot`).
277             """
278             super().__init__(*args, **kwargs)
279
280             self.point_i: Tuple[float, float] = (1., 0.)
281             self.point_j: Tuple[float, float] = (0., 1.)
282
283         @property
284         def _matrix(self) -> MatrixType:
285             """Return the assembled matrix of the basis vectors."""
286             return np.array([
287                 [self.point_i[0], self.point_j[0]],
288                 [self.point_i[1], self.point_j[1]]
289             ])
290
291         @property
292         def _det(self) -> float:
293             """Return the determinant of the assembled matrix."""
294             return float(np.linalg.det(self._matrix))
295
296         @property
297         def _eigs(self) -> 'Iterable[Tuple[float, VectorType]]':
298             """Return the eigenvalues and eigenvectors zipped together to be iterated over.
299
300             :rtype: Iterable[Tuple[float, VectorType]]
301             """
302             values, vectors = np.linalg.eig(self._matrix)
303             return zip(values, vectors.T)
304
305         @abstractmethod
306         def paintEvent(self, event: QPaintEvent) -> None:
307             """Handle a :class:`QPaintEvent`."""
308
309         def _draw_parallel_lines(self, painter: QPainter, vector: Tuple[float, float], point: Tuple[float, float]) ->
            ↪  None:
310             """Draw a set of evenly spaced grid lines parallel to ``vector`` intersecting ``point``.
311
312             :param QPainter painter: The painter to draw the lines with
313             :param vector: The vector to draw the grid lines parallel to
314             :type vector: Tuple[float, float]
315             :param point: The point for the lines to intersect with
316             :type point: Tuple[float, float]
317             """
318             max_x, max_y = self._grid_corner()
319             vector_x, vector_y = vector
320             point_x, point_y = point
321
322             # If the determinant is 0
323             if abs(vector_x * point_y - vector_y * point_x) < 1e-12:
324                 rank = np.linalg.matrix_rank(
```

```
                np.array([
                    [vector_x, point_x],
                    [vector_y, point_y]
                ])
            )

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

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

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

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

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

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

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

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

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

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

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

            self._draw_oblique_line(painter, m, 0)
```

```
398
399             # We don't want to overshoot the max number of parallel lines,
400             # but we should also stop looping as soon as we can't draw any more lines
401             for i in range(1, self._MAX_PARALLEL_LINES + 1):
402                 if not self._draw_pair_of_oblique_lines(painter, m, i * c):
403                     break
404
405     def _draw_pair_of_oblique_lines(self, painter: QPainter, m: float, c: float) -> bool:
406         """Draw a pair of oblique lines, using the equation y = mx + c.
407
408         This method just calls :meth:`_draw_oblique_line` with ``c`` and ``-c``,
409         and returns True if either call returned True.
410
411         :param QPainter painter: The painter to draw the vectors and grid lines with
412         :param float m: The gradient of the lines to draw
413         :param float c: The y-intercept of the lines to draw. We use the positive and negative versions
414         :returns bool: Whether we were able to draw any lines on the canvas
415         """
416         return any([
417             self._draw_oblique_line(painter, m, c),
418             self._draw_oblique_line(painter, m, -c)
419         ])
420
421     def _draw_oblique_line(self, painter: QPainter, m: float, c: float) -> bool:
422         """Draw an oblique line, using the equation y = mx + c.
423
424         We only draw the part of the line that fits within the canvas, returning True if
425         we were able to draw a line within the boundaries, and False if we couldn't draw a line
426
427         :param QPainter painter: The painter to draw the vectors and grid lines with
428         :param float m: The gradient of the line to draw
429         :param float c: The y-intercept of the line to draw
430         :returns bool: Whether we were able to draw a line on the canvas
431         """
432         max_x, max_y = self._grid_corner()
433
434         # These variable names are shortened for convenience
435         # myi is max_y_intersection, mmyi is minus_max_y_intersection, etc.
436         myi = (max_y - c) / m
437         mmyi = (-max_y - c) / m
438         mxi = max_x * m + c
439         mmxi = -max_x * m + c
440
441         # The inner list here is a list of coords, or None
442         # If an intersection fits within the bounds, then we keep its coord,
443         # else it is None, and then gets discarded from the points list
444         # By the end, points is a list of two coords, or an empty list
445         points: List[Tuple[float, float]] = [
446             x for x in [
447                 (myi, max_y) if -max_x < myi < max_x else None,
448                 (mmyi, -max_y) if -max_x < mmyi < max_x else None,
449                 (max_x, mxi) if -max_y < mxi < max_y else None,
450                 (-max_x, mmxi) if -max_y < mmxi < max_y else None
451             ] if x is not None
452         ]
453
454         # If no intersections fit on the canvas
455         if len(points) < 2:
456             return False
457
458         # If we can, then draw the line
459         else:
460             painter.drawLine(
461                 *self.canvas_coords(*points[0]),
462                 *self.canvas_coords(*points[1])
463             )
464             return True
465
466     def _draw_transformed_grid(self, painter: QPainter) -> None:
467         """Draw the transformed version of the grid, given by the basis vectors.
468
469         .. note:: This method draws the grid, but not the basis vectors. Use :meth:`_draw_basis_vectors` to draw
↪  them.
```

```
470
471             :param QPainter painter: The painter to draw the grid lines with
472             """
473             # Draw all the parallel lines
474             painter.setPen(QPen(self._COLOUR_I, self._WIDTH_TRANSFORMED_GRID))
475             self._draw_parallel_lines(painter, self.point_i, self.point_j)
476             painter.setPen(QPen(self._COLOUR_J, self._WIDTH_TRANSFORMED_GRID))
477             self._draw_parallel_lines(painter, self.point_j, self.point_i)
478
479         def _draw_arrowhead_away_from_origin(self, painter: QPainter, point: Tuple[float, float]) -> None:
480             """Draw an arrowhead at ``point``, pointing away from the origin.
481
482             :param QPainter painter: The painter to draw the arrowhead with
483             :param point: The point to draw the arrowhead at, given in grid coords
484             :type point: Tuple[float, float]
485             """
486             # This algorithm was adapted from a C# algorithm found at
487             # http://csharphelper.com/blog/2014/12/draw-lines-with-arrowheads-in-c/
488
489             # Get the x and y coords of the point, and then normalize them
490             # We have to normalize them, or else the size of the arrowhead will
491             # scale with the distance of the point from the origin
492             x, y = point
493             vector_length = np.sqrt(x * x + y * y)
494
495             if vector_length < 1e-12:
496                 return
497
498             nx = x / vector_length
499             ny = y / vector_length
500
501             # We choose a length and find the steps in the x and y directions
502             length = min(
503                 self._ARROWHEAD_LENGTH * self.DEFAULT_GRID_SPACING / self.grid_spacing,
504                 vector_length
505             )
506             dx = length * (-nx - ny)
507             dy = length * (nx - ny)
508
509             # Then we just plot those lines
510             painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x + dx, y + dy))
511             painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x - dy, y + dx))
512
513         def _draw_position_vector(self, painter: QPainter, point: Tuple[float, float], colour: QColor) -> None:
514             """Draw a vector from the origin to the given point.
515
516             :param QPainter painter: The painter to draw the position vector with
517             :param point: The tip of the position vector in grid coords
518             :type point: Tuple[float, float]
519             :param QColor colour: The colour to draw the position vector in
520             """
521             painter.setPen(QPen(colour, self._WIDTH_VECTOR_LINE))
522             painter.drawLine(*self._canvas_origin, *self.canvas_coords(*point))
523             self._draw_arrowhead_away_from_origin(painter, point)
524
525         def _draw_basis_vectors(self, painter: QPainter) -> None:
526             """Draw arrowheads at the tips of the basis vectors.
527
528             :param QPainter painter: The painter to draw the basis vectors with
529             """
530             self._draw_position_vector(painter, self.point_i, self._COLOUR_I)
531             self._draw_position_vector(painter, self.point_j, self._COLOUR_J)
532
533         def _draw_basis_vector_labels(self, painter: QPainter) -> None:
534             """Label the basis vectors with `i` and `j`."""
535             font = self.font()
536             font.setItalic(True)
537             font.setStyleHint(QFont.Serif)
538
539             self._draw_text_at_vector_tip(painter, self.point_i, 'i', font)
540             self._draw_text_at_vector_tip(painter, self.point_j, 'j', font)
541
542         def _draw_text_at_vector_tip(
```

```
            self,
            painter: QPainter,
            point: Tuple[float, float],
            text: str,
            font: Optional[QFont] = None
        ) -> None:
            """Draw the given text at the point as if it were the tip of a vector, using the custom font if given."""
            offset = 3
            top_left: QPoint
            bottom_right: QPoint
            alignment_flags: int
            x, y = point

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

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

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

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

            original_font = painter.font()

            if font is not None:
                painter.setFont(font)

            painter.setPen(QPen(self._COLOUR_TEXT, 1))
            painter.drawText(QRectF(top_left, bottom_right), alignment_flags, text)

            painter.setFont(original_font)


    class VisualizeTransformationPlot(VectorGridPlot):
        """This class is a superclass for visualizing transformations. It provides utility methods."""

        _COLOUR_EIGEN = QColor('#13cf00')
        """This is the colour of the eigenvectors and eigenlines (the spans of the eigenvectors)."""

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

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

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

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

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

            painter.fillPath(path, brush)
```

```
616         def _draw_determinant_text(self, painter: QPainter) -> None:
617             """Write the string value of the determinant in the middle of the parallelogram.
618 
619             :param QPainter painter: The painter to draw the determinant text with
620             """
621             painter.setPen(QPen(self._COLOUR_TEXT, self._WIDTH_VECTOR_LINE))
622 
623             # We're building a QRect that encloses the determinant parallelogram
624             # Then we can center the text in this QRect
625             coords: List[Tuple[float, float]] = [
626                 (0, 0),
627                 self.point_i,
628                 self.point_j,
629                 (
630                     self.point_i[0] + self.point_j[0],
631                     self.point_i[1] + self.point_j[1]
632                 )
633             ]
634 
635             xs = [t[0] for t in coords]
636             ys = [t[1] for t in coords]
637 
638             top_left = QPoint(*self.canvas_coords(min(xs), max(ys)))
639             bottom_right = QPoint(*self.canvas_coords(max(xs), min(ys)))
640 
641             rect = QRectF(top_left, bottom_right)
642 
643             painter.drawText(
644                 rect,
645                 Qt.AlignHCenter | Qt.AlignVCenter,
646                 f'{self._det:.2f}'
647             )
648 
649         def _draw_eigenvectors(self, painter: QPainter) -> None:
650             """Draw the eigenvectors of the displayed matrix transformation.
651 
652             :param QPainter painter: The painter to draw the eigenvectors with
653             """
654             for value, vector in self._eigs:
655                 x = value * vector[0]
656                 y = value * vector[1]
657 
658                 if x.imag != 0 or y.imag != 0:
659                     continue
660 
661                 self._draw_position_vector(painter, (x, y), self._COLOUR_EIGEN)
662                 self._draw_text_at_vector_tip(painter, (x, y), f'{value:.2f}')
663 
664         def _draw_eigenlines(self, painter: QPainter) -> None:
665             """Draw the eigenlines. These are the invariant lines, or the spans of the eigenvectors.
666 
667             :param QPainter painter: The painter to draw the eigenlines with
668             """
669             painter.setPen(QPen(self._COLOUR_EIGEN, self._WIDTH_TRANSFORMED_GRID))
670 
671             for value, vector in self._eigs:
672                 if value.imag != 0:
673                     continue
674 
675                 x, y = vector
676 
677                 if x == 0:
678                     x_mid = int(self.width() / 2)
679                     painter.drawLine(x_mid, 0, x_mid, self.height())
680 
681                 elif y == 0:
682                     y_mid = int(self.height() / 2)
683                     painter.drawLine(0, y_mid, self.width(), y_mid)
684 
685                 else:
686                     self._draw_oblique_line(painter, y / x, 0)
687 
688         def _draw_polygon_from_points(self, painter: QPainter, points: List[Tuple[float, float]]) -> None:
```

```
689             """Draw a polygon from a given list of points.
690
691             This is a helper method for :meth:`_draw_untransformed_polygon` and :meth:`_draw_transformed_polygon`.
692             """
693             if len(points) > 2:
694                 painter.drawPolygon(QPolygonF(
695                     [QPointF(*self.canvas_coords(*p)) for p in points]
696                 ))
697             elif len(points) == 2:
698                 painter.drawLine(
699                     *self.canvas_coords(*points[0]),
700                     *self.canvas_coords(*points[1])
701                 )
702
703         def _draw_untransformed_polygon(self, painter: QPainter) -> None:
704             """Draw the original untransformed polygon with a dashed line."""
705             pen = QPen(self._PEN_POLYGON)
706             pen.setDashPattern([4, 4])
707             painter.setPen(pen)
708
709             self._draw_polygon_from_points(painter, self.polygon_points)
710
711         def _draw_transformed_polygon(self, painter: QPainter) -> None:
712             """Draw the transformed version of the polygon."""
713             if len(self.polygon_points) == 0:
714                 return
715
716             painter.setPen(self._PEN_POLYGON)
717
718             # This transpose trick lets us do one matrix multiplication to transform every point in the polygon
719             # I learned this from Phil. Thanks Phil
720             self._draw_polygon_from_points(
721                 painter,
722                 (self._matrix @ np.array(self.polygon_points).T).T
723             )
```

A.14 gui/plots/__init__.py

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

A.15 gui/dialogs/misc.py

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

```
13  from typing import Dict, List, Optional, Tuple, Union
14
15  from PyQt5.QtCore import PYQT_VERSION_STR, QT_VERSION_STR, Qt, pyqtSlot
16  from PyQt5.QtGui import QKeySequence
17  from PyQt5.QtWidgets import (QDialog, QFileDialog, QGridLayout, QGroupBox,
18                               QHBoxLayout, QLabel, QPushButton, QRadioButton,
19                               QShortcut, QSizePolicy, QSpacerItem,
20                               QStackedLayout, QVBoxLayout, QWidget)
21
22  import lintrans
23  from lintrans.global_settings import GlobalSettings
24  from lintrans.gui.plots import DefinePolygonWidget
25  from lintrans.matrices import MatrixWrapper
26  from lintrans.matrices.utility import round_float
27  from lintrans.typing_ import MatrixType, is_matrix_type
28  from lintrans.updating import update_lintrans_in_background
29
30
31  class FixedSizeDialog(QDialog):
32      """A simple superclass to create modal dialog boxes with fixed size.
33
34      We override the :meth:`open` method to set the fixed size as soon as the dialog is opened modally.
35      """
36
37      def __init__(self, *args, **kwargs) -> None:
38          """Set the :cpp:enum:`Qt::WA_DeleteOnClose` attribute to ensure deletion of dialog."""
39          super().__init__(*args, **kwargs)
40          self.setAttribute(Qt.WA_DeleteOnClose)
41          self.setWindowFlag(Qt.WindowContextHelpButtonHint, False)
42
43      def open(self) -> None:
44          """Override :meth:`QDialog.open` to set the dialog to a fixed size."""
45          super().open()
46          self.setFixedSize(self.size())
47
48
49  class AboutDialog(FixedSizeDialog):
50      """A simple dialog class to display information about the app to the user.
51
52      It only has an :meth:`__init__` method because it only has label widgets, so no other methods are necessary
    ↪ here.
53      """
54
55      def __init__(self, *args, **kwargs):
56          """Create an :class:`AboutDialog` object with all the label widgets."""
57          super().__init__(*args, **kwargs)
58
59          self.setWindowTitle('About lintrans')
60
61          # === Create the widgets
62
63          label_title = QLabel(self)
64          label_title.setText(f'lintrans (version {lintrans.__version__})')
65          label_title.setAlignment(Qt.AlignCenter)
66
67          font_title = label_title.font()
68          font_title.setPointSize(font_title.pointSize() * 2)
69          label_title.setFont(font_title)
70
71          label_version_info = QLabel(self)
72          label_version_info.setText(
73              f'With Python version {platform.python_version()}\n'
74              f'Qt version {QT_VERSION_STR} and PyQt5 version {PYQT_VERSION_STR}\n'
75              f'Running on {platform.platform()}'
76          )
77          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):
224                 # tl = top left, br = bottom right, etc.
225                 label_tl = QLabel(self)
226                 label_tl.setText(round_float(matrix[0][0]))
227 
228                 label_tr = QLabel(self)
229                 label_tr.setText(round_float(matrix[0][1]))
230 
```

```
231                 label_bl = QLabel(self)
232                 label_bl.setText(round_float(matrix[1][0]))
233
234                 label_br = QLabel(self)
235                 label_br.setText(round_float(matrix[1][1]))
236
237                 # The parens need to be bigger than the numbers, but increasing the font size also
238                 # makes the font thicker, so we have to reduce the font weight by the same factor
239                 font_parens = self.font()
240                 font_parens.setPointSize(int(font_parens.pointSize() * 2.5))
241                 font_parens.setWeight(int(font_parens.weight() / 2.5))
242
243                 label_paren_left = QLabel(self)
244                 label_paren_left.setText('(')
245                 label_paren_left.setFont(font_parens)
246
247                 label_paren_right = QLabel(self)
248                 label_paren_right.setText(')')
249                 label_paren_right.setFont(font_parens)
250
251                 container = QWidget(self)
252                 grid_layout = QGridLayout()
253
254                 grid_layout.addWidget(label_paren_left, 0, 0, -1, 1)
255                 grid_layout.addWidget(label_tl, 0, 1)
256                 grid_layout.addWidget(label_tr, 0, 2)
257                 grid_layout.addWidget(label_bl, 1, 1)
258                 grid_layout.addWidget(label_br, 1, 2)
259                 grid_layout.addWidget(label_paren_right, 0, 3, -1, 1)
260
261                 container.setLayout(grid_layout)
262
263                 return container
264
265         raise ValueError('Matrix was not MatrixType or str')
266
267
268 class FileSelectDialog(QFileDialog):
269     """A subclass of :class:`QFileDialog` that fixes an issue with the default suffix on UNIX platforms."""
270
271     def selectedFiles(self) -> List[str]:
272         """Return a list of strings containing the absolute paths of the selected files in the dialog.
273
274         There is an issue on UNIX platforms where a hidden directory will be recognised as a suffix.
275         For example, ``/home/dyson/.lintrans/saves/test`` should have ``.lt`` appended, but
276         ``.lintrans/saves/test`` gets recognised as the suffix, so the default suffix is not added.
277
278         To fix this, we just look at the basename and see if it needs a suffix added. We do this for
279         every name in the list, but there should be just one name, since this class is only intended
280         to be used for saving files. We still return the full list of filenames.
281         """
282         selected_files: List[str] = []
283
284         for filename in super().selectedFiles():
285             # path will be the full path of the file, without the extension
286             # This method understands hidden directories on UNIX platforms
287             path, ext = os.path.splitext(filename)
288
289             if ext == '':
290                 ext = '.' + self.defaultSuffix()
291
292             selected_files.append(''.join((path, ext)))
293
294         return selected_files
295
296
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                 f'You can change your choice at any time by editing {GlobalSettings().get_settings_file()}'
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(gs.UpdateType.auto)
438
439             elif self._radio_button_prompt.isChecked():
440                 gs.set_update_type(gs.UpdateType.prompt)
441
442             elif self._radio_button_never.isChecked():
443                 gs.set_update_type(gs.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.16 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>
6
7   """This package provides separate dialogs for the main GUI.
8
9   These dialogs are for defining new matrices in different ways and editing settings.
10  """
11
12  from .define_new_matrix import (DefineAsExpressionDialog, DefineMatrixDialog,
13                                  DefineNumericallyDialog, DefineVisuallyDialog)
14  from .misc import (AboutDialog, DefinePolygonDialog, FileSelectDialog,
15                     InfoPanelDialog, PromptUpdateDialog)
16  from .settings import DisplaySettingsDialog
17
18  __all__ = ['AboutDialog', 'DefineAsExpressionDialog', 'DefineMatrixDialog',
19             'DefineNumericallyDialog', 'DefinePolygonDialog', 'DefineVisuallyDialog',
20             'DisplaySettingsDialog', 'FileSelectDialog', 'InfoPanelDialog', 'PromptUpdateDialog']
```

A.17 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 QIntValidator, QKeyEvent, QKeySequence
17  from PyQt5.QtWidgets import (QCheckBox, QGroupBox, QHBoxLayout, QLayout,
18                               QShortcut, QSizePolicy, QSpacerItem, QVBoxLayout)
19
20  from lintrans.gui.dialogs.misc import FixedSizeDialog
21  from lintrans.gui.settings import DisplaySettings
22
23
24  class SettingsDialog(FixedSizeDialog):
25      """An abstract superclass for other simple dialogs."""
26
27      def __init__(self, *args, resettable: bool, **kwargs):
28          """Create the widgets and layout of the dialog, passing ``*args`` and ``**kwargs`` to super."""
29          super().__init__(*args, **kwargs)
30
31          # === Create the widgets
32
33          self._button_confirm = QtWidgets.QPushButton(self)
34          self._button_confirm.setText('Confirm')
35          self._button_confirm.clicked.connect(self._confirm_settings)
36          self._button_confirm.setToolTip('Confirm these new settings<br><b>(Ctrl + Enter)</b>')
37          QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self._button_confirm.click)
38
39          self._button_cancel = QtWidgets.QPushButton(self)
40          self._button_cancel.setText('Cancel')
41          self._button_cancel.clicked.connect(self.reject)
```

```
42              self._button_cancel.setToolTip('Revert these settings<br><b>(Escape)</b>')
43
44              if resettable:
45                  self._button_reset = QtWidgets.QPushButton(self)
46                  self._button_reset.setText('Reset to defaults')
47                  self._button_reset.clicked.connect(self._reset_settings)
48                  self._button_reset.setToolTip('Reset these settings to their defaults<br><b>(Ctrl + R)</b>')
49                  QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(self._button_reset.click)
50
51              # === Arrange the widgets
52
53              self.setContentsMargins(10, 10, 10, 10)
54
55              self._hlay_buttons = QHBoxLayout()
56              self._hlay_buttons.setSpacing(20)
57
58              if resettable:
59                  self._hlay_buttons.addWidget(self._button_reset)
60
61              self._hlay_buttons.addItem(QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum))
62              self._hlay_buttons.addWidget(self._button_cancel)
63              self._hlay_buttons.addWidget(self._button_confirm)
64
65          def _setup_layout(self, options_layout: QLayout) -> None:
66              """Set the layout of the settings widget.
67
68              .. note:: This method must be called at the end of :meth:`__init__`
69                 in subclasses to setup the layout properly.
70              """
71              vlay_all = QVBoxLayout()
72              vlay_all.setSpacing(20)
73              vlay_all.addLayout(options_layout)
74              vlay_all.addLayout(self._hlay_buttons)
75
76              self.setLayout(vlay_all)
77
78          @abc.abstractmethod
79          def _load_settings(self) -> None:
80              """Load the current settings into the widgets."""
81
82          @abc.abstractmethod
83          def _confirm_settings(self) -> None:
84              """Confirm the settings chosen in the dialog."""
85
86          def _reset_settings(self) -> None:
87              """Reset the settings.
88
89              .. note:: This method is empty but not abstract because not all subclasses will need to implement it.
90              """
91
92
93      class DisplaySettingsDialog(SettingsDialog):
94          """The dialog to allow the user to edit the display settings."""
95
96          def __init__(self, *args, display_settings: DisplaySettings, **kwargs):
97              """Create the widgets and layout of the dialog.
98
99              :param DisplaySettings display_settings: The :class:`~lintrans.gui.settings.DisplaySettings` object to
    ↪  mutate
100             """
101             super().__init__(*args, resettable=True, **kwargs)
102
103             self.display_settings = display_settings
104             self.setWindowTitle('Change display settings')
105
106             self._dict_checkboxes: Dict[str, QCheckBox] = {}
107
108             # === Create the widgets
109
110             # Basic stuff
111
112             self._checkbox_draw_background_grid = QCheckBox(self)
113             self._checkbox_draw_background_grid.setText('Draw &background grid')
```

```
114             self._checkbox_draw_background_grid.setToolTip(
115                 'Draw the background grid (axes are always drawn)'
116             )
117             self._dict_checkboxes['b'] = self._checkbox_draw_background_grid
118
119             self._checkbox_draw_transformed_grid = QCheckBox(self)
120             self._checkbox_draw_transformed_grid.setText('Draw t&ransformed grid')
121             self._checkbox_draw_transformed_grid.setToolTip(
122                 'Draw the transformed grid (vectors are handled separately)'
123             )
124             self._dict_checkboxes['r'] = self._checkbox_draw_transformed_grid
125
126             self._checkbox_draw_basis_vectors = QCheckBox(self)
127             self._checkbox_draw_basis_vectors.setText('Draw basis &vectors')
128             self._checkbox_draw_basis_vectors.setToolTip(
129                 'Draw the transformed basis vectors'
130             )
131             self._checkbox_draw_basis_vectors.clicked.connect(self._update_gui)
132             self._dict_checkboxes['v'] = self._checkbox_draw_basis_vectors
133
134             self._checkbox_label_basis_vectors = QCheckBox(self)
135             self._checkbox_label_basis_vectors.setText('Label the bas&is vectors')
136             self._checkbox_label_basis_vectors.setToolTip(
137                 'Label the transformed i and j basis vectors'
138             )
139             self._dict_checkboxes['i'] = self._checkbox_label_basis_vectors
140
141             # Animations
142
143             self._checkbox_smoothen_determinant = QCheckBox(self)
144             self._checkbox_smoothen_determinant.setText('&Smoothen determinant')
145             self._checkbox_smoothen_determinant.setToolTip(
146                 'Smoothly animate the determinant transition during animation (if possible)'
147             )
148             self._dict_checkboxes['s'] = self._checkbox_smoothen_determinant
149
150             self._checkbox_applicative_animation = QCheckBox(self)
151             self._checkbox_applicative_animation.setText('&Applicative animation')
152             self._checkbox_applicative_animation.setToolTip(
153                 'Animate the new transformation applied to the current one,\n'
154                 'rather than just that transformation on its own'
155             )
156             self._dict_checkboxes['a'] = self._checkbox_applicative_animation
157
158             label_animation_time = QtWidgets.QLabel(self)
159             label_animation_time.setText('Total animation length (ms)')
160             label_animation_time.setToolTip(
161                 'How long it takes for an animation to complete'
162             )
163
164             self._lineedit_animation_time = QtWidgets.QLineEdit(self)
165             self._lineedit_animation_time.setValidator(QIntValidator(1, 9999, self))
166             self._lineedit_animation_time.textChanged.connect(self._update_gui)
167
168             label_animation_pause_length = QtWidgets.QLabel(self)
169             label_animation_pause_length.setText('Animation pause length (ms)')
170             label_animation_pause_length.setToolTip(
171                 'How many milliseconds to pause for in comma-separated animations'
172             )
173
174             self._lineedit_animation_pause_length = QtWidgets.QLineEdit(self)
175             self._lineedit_animation_pause_length.setValidator(QIntValidator(1, 999, self))
176
177             # Matrix info
178
179             self._checkbox_draw_determinant_parallelogram = QCheckBox(self)
180             self._checkbox_draw_determinant_parallelogram.setText('Draw &determinant parallelogram')
181             self._checkbox_draw_determinant_parallelogram.setToolTip(
182                 'Shade the parallelogram representing the determinant of the matrix'
183             )
184             self._checkbox_draw_determinant_parallelogram.clicked.connect(self._update_gui)
185             self._dict_checkboxes['d'] = self._checkbox_draw_determinant_parallelogram
186
```

```
187             self._checkbox_show_determinant_value = QCheckBox(self)
188             self._checkbox_show_determinant_value.setText('Show de&terminant value')
189             self._checkbox_show_determinant_value.setToolTip(
190                 'Show the value of the determinant inside the parallelogram'
191             )
192             self._dict_checkboxes['t'] = self._checkbox_show_determinant_value
193
194             self._checkbox_draw_eigenvectors = QCheckBox(self)
195             self._checkbox_draw_eigenvectors.setText('Draw &eigenvectors')
196             self._checkbox_draw_eigenvectors.setToolTip('Draw the eigenvectors of the transformations')
197             self._dict_checkboxes['e'] = self._checkbox_draw_eigenvectors
198
199             self._checkbox_draw_eigenlines = QCheckBox(self)
200             self._checkbox_draw_eigenlines.setText('Draw eigen&lines')
201             self._checkbox_draw_eigenlines.setToolTip('Draw the eigenlines (invariant lines) of the transformations')
202             self._dict_checkboxes['l'] = self._checkbox_draw_eigenlines
203
204             # Polygon
205
206             self._checkbox_draw_untransformed_polygon = QCheckBox(self)
207             self._checkbox_draw_untransformed_polygon.setText('&Untransformed polygon')
208             self._checkbox_draw_untransformed_polygon.setToolTip('Draw the untransformed version of the polygon')
209             self._dict_checkboxes['u'] = self._checkbox_draw_untransformed_polygon
210
211             self._checkbox_draw_transformed_polygon = QCheckBox(self)
212             self._checkbox_draw_transformed_polygon.setText('Transformed &polygon')
213             self._checkbox_draw_transformed_polygon.setToolTip('Draw the transformed version of the polygon')
214             self._dict_checkboxes['p'] = self._checkbox_draw_transformed_polygon
215
216             # Input/output vectors
217
218             self._checkbox_draw_input_vector = QCheckBox(self)
219             self._checkbox_draw_input_vector.setText('Draw the i&nput vector')
220             self._checkbox_draw_input_vector.setToolTip('Draw the input vector (only in the viewport)')
221             self._dict_checkboxes['n'] = self._checkbox_draw_input_vector
222
223             self._checkbox_draw_output_vector = QCheckBox(self)
224             self._checkbox_draw_output_vector.setText('Draw the &output vector')
225             self._checkbox_draw_output_vector.setToolTip('Draw the output vector (only in the viewport)')
226             self._dict_checkboxes['o'] = self._checkbox_draw_output_vector
227
228             # === Arrange the widgets in QGroupBoxes
229
230             # Basic stuff
231
232             vlay_groupbox_basic_stuff = QVBoxLayout()
233             vlay_groupbox_basic_stuff.setSpacing(20)
234             vlay_groupbox_basic_stuff.addWidget(self._checkbox_draw_background_grid)
235             vlay_groupbox_basic_stuff.addWidget(self._checkbox_draw_transformed_grid)
236             vlay_groupbox_basic_stuff.addWidget(self._checkbox_draw_basis_vectors)
237             vlay_groupbox_basic_stuff.addWidget(self._checkbox_label_basis_vectors)
238
239             groupbox_basic_stuff = QGroupBox('Basic stuff', self)
240             groupbox_basic_stuff.setLayout(vlay_groupbox_basic_stuff)
241
242             # Animations
243
244             hlay_animation_time = QHBoxLayout()
245             hlay_animation_time.addWidget(label_animation_time)
246             hlay_animation_time.addWidget(self._lineedit_animation_time)
247
248             hlay_animation_pause_length = QHBoxLayout()
249             hlay_animation_pause_length.addWidget(label_animation_pause_length)
250             hlay_animation_pause_length.addWidget(self._lineedit_animation_pause_length)
251
252             vlay_groupbox_animations = QVBoxLayout()
253             vlay_groupbox_animations.setSpacing(20)
254             vlay_groupbox_animations.addWidget(self._checkbox_smoothen_determinant)
255             vlay_groupbox_animations.addWidget(self._checkbox_applicative_animation)
256             vlay_groupbox_animations.addLayout(hlay_animation_time)
257             vlay_groupbox_animations.addLayout(hlay_animation_pause_length)
258
259             groupbox_animations = QGroupBox('Animations', self)
```

```
260             groupbox_animations.setLayout(vlay_groupbox_animations)
261
262             # Matrix info
263
264             vlay_groupbox_matrix_info = QVBoxLayout()
265             vlay_groupbox_matrix_info.setSpacing(20)
266             vlay_groupbox_matrix_info.addWidget(self._checkbox_draw_determinant_parallelogram)
267             vlay_groupbox_matrix_info.addWidget(self._checkbox_show_determinant_value)
268             vlay_groupbox_matrix_info.addWidget(self._checkbox_draw_eigenvectors)
269             vlay_groupbox_matrix_info.addWidget(self._checkbox_draw_eigenlines)
270
271             groupbox_matrix_info = QGroupBox('Matrix info', self)
272             groupbox_matrix_info.setLayout(vlay_groupbox_matrix_info)
273
274             # Polygon
275
276             vlay_groupbox_polygon = QVBoxLayout()
277             vlay_groupbox_polygon.setSpacing(20)
278             vlay_groupbox_polygon.addWidget(self._checkbox_draw_untransformed_polygon)
279             vlay_groupbox_polygon.addWidget(self._checkbox_draw_transformed_polygon)
280
281             groupbox_polygon = QGroupBox('Polygon', self)
282             groupbox_polygon.setLayout(vlay_groupbox_polygon)
283
284             # Input/output vectors
285
286             vlay_groupbox_io_vectors = QVBoxLayout()
287             vlay_groupbox_io_vectors.setSpacing(20)
288             vlay_groupbox_io_vectors.addWidget(self._checkbox_draw_input_vector)
289             vlay_groupbox_io_vectors.addWidget(self._checkbox_draw_output_vector)
290
291             groupbox_io_vectors = QGroupBox('Input/output vectors', self)
292             groupbox_io_vectors.setLayout(vlay_groupbox_io_vectors)
293
294             # Now arrange the groupboxes
295             vlay_left = QVBoxLayout()
296             vlay_left.setSpacing(20)
297             vlay_left.addWidget(groupbox_basic_stuff)
298             vlay_left.addWidget(groupbox_animations)
299
300             vlay_right = QVBoxLayout()
301             vlay_right.setSpacing(20)
302             vlay_right.addWidget(groupbox_matrix_info)
303             vlay_right.addWidget(groupbox_polygon)
304             vlay_right.addWidget(groupbox_io_vectors)
305
306             options_layout = QHBoxLayout()
307             options_layout.setSpacing(20)
308             options_layout.addLayout(vlay_left)
309             options_layout.addLayout(vlay_right)
310
311             self._setup_layout(options_layout)
312
313             # Finally, we load the current settings and update the GUI
314             self._load_settings()
315             self._update_gui()
316
317         def _load_settings(self) -> None:
318             """Load the current display settings into the widgets."""
319             # Basic stuff
320             self._checkbox_draw_background_grid.setChecked(self.display_settings.draw_background_grid)
321             self._checkbox_draw_transformed_grid.setChecked(self.display_settings.draw_transformed_grid)
322             self._checkbox_draw_basis_vectors.setChecked(self.display_settings.draw_basis_vectors)
323             self._checkbox_label_basis_vectors.setChecked(self.display_settings.label_basis_vectors)
324
325             # Animations
326             self._checkbox_smoothen_determinant.setChecked(self.display_settings.smoothen_determinant)
327             self._checkbox_applicative_animation.setChecked(self.display_settings.applicative_animation)
328             self._lineedit_animation_time.setText(str(self.display_settings.animation_time))
329             self._lineedit_animation_pause_length.setText(str(self.display_settings.animation_pause_length))
330
331             # Matrix info
```

```
332             self._checkbox_draw_determinant_parallelogram.setChecked(
                ↪  self.display_settings.draw_determinant_parallelogram)
333             self._checkbox_show_determinant_value.setChecked(self.display_settings.show_determinant_value)
334             self._checkbox_draw_eigenvectors.setChecked(self.display_settings.draw_eigenvectors)
335             self._checkbox_draw_eigenlines.setChecked(self.display_settings.draw_eigenlines)
336
337             # Polygon
338             self._checkbox_draw_untransformed_polygon.setChecked(self.display_settings.draw_untransformed_polygon)
339             self._checkbox_draw_transformed_polygon.setChecked(self.display_settings.draw_transformed_polygon)
340
341             # Input/output vectors
342             self._checkbox_draw_input_vector.setChecked(self.display_settings.draw_input_vector)
343             self._checkbox_draw_output_vector.setChecked(self.display_settings.draw_output_vector)
344
345         def _confirm_settings(self) -> None:
346             """Build a :class:`~lintrans.gui.settings.DisplaySettings` object and assign it."""
347             # Basic stuff
348             self.display_settings.draw_background_grid = self._checkbox_draw_background_grid.isChecked()
349             self.display_settings.draw_transformed_grid = self._checkbox_draw_transformed_grid.isChecked()
350             self.display_settings.draw_basis_vectors = self._checkbox_draw_basis_vectors.isChecked()
351             self.display_settings.label_basis_vectors = self._checkbox_label_basis_vectors.isChecked()
352
353             # Animations
354             self.display_settings.smoothen_determinant = self._checkbox_smoothen_determinant.isChecked()
355             self.display_settings.applicative_animation = self._checkbox_applicative_animation.isChecked()
356             self.display_settings.animation_time = int(self._lineedit_animation_time.text())
357             self.display_settings.animation_pause_length = int(self._lineedit_animation_pause_length.text())
358
359             # Matrix info
360             self.display_settings.draw_determinant_parallelogram =
                ↪  self._checkbox_draw_determinant_parallelogram.isChecked()
361             self.display_settings.show_determinant_value = self._checkbox_show_determinant_value.isChecked()
362             self.display_settings.draw_eigenvectors = self._checkbox_draw_eigenvectors.isChecked()
363             self.display_settings.draw_eigenlines = self._checkbox_draw_eigenlines.isChecked()
364
365             # Polygon
366             self.display_settings.draw_untransformed_polygon = self._checkbox_draw_untransformed_polygon.isChecked()
367             self.display_settings.draw_transformed_polygon = self._checkbox_draw_transformed_polygon.isChecked()
368
369             # Input/output vectors
370             self.display_settings.draw_input_vector = self._checkbox_draw_input_vector.isChecked()
371             self.display_settings.draw_output_vector = self._checkbox_draw_output_vector.isChecked()
372
373             self.accept()
374
375         def _reset_settings(self) -> None:
376             """Reset the display settings to their defaults."""
377             self.display_settings = DisplaySettings()
378             self._load_settings()
379             self._update_gui()
380
381         def _update_gui(self) -> None:
382             """Update the GUI according to other widgets in the GUI.
383
384             For example, this method updates which checkboxes are enabled based on the values of other checkboxes.
385             """
386             self._checkbox_show_determinant_value.setEnabled(self._checkbox_draw_determinant_parallelogram.isChecked())
387             self._checkbox_label_basis_vectors.setEnabled(self._checkbox_draw_basis_vectors.isChecked())
388
389             try:
390                 self._button_confirm.setEnabled(int(self._lineedit_animation_time.text()) >= 10)
391             except ValueError:
392                 self._button_confirm.setEnabled(False)
393
394         def keyPressEvent(self, event: QKeyEvent) -> None:
395             """Handle a :class:`QKeyEvent` by manually activating toggling checkboxes.
396
397             Qt handles these shortcuts automatically and allows the user to do ``Alt + Key``
398             to activate a simple shortcut defined with ``&``. However, I like to be able to
399             just hit ``Key`` and have the shortcut activate.
400             """
401             letter = event.text().lower()
402             key = event.key()
```

```
403
404             if letter in self._dict_checkboxes:
405                 self._dict_checkboxes[letter].animateClick()
406
407             # Return or keypad enter
408             elif key == Qt.Key_Return or key == Qt.Key_Enter:
409                 self._button_confirm.click()
410
411             # Escape
412             elif key == Qt.Key_Escape:
413                 self._button_cancel.click()
414
415             else:
416                 event.ignore()
417                 return
418
419             event.accept()
```

A.18 gui/dialogs/define_new_matrix.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 an abstract :class:`DefineMatrixDialog` class and subclasses."""
8
9   from __future__ import annotations
10
11  import abc
12  from typing import List, Tuple
13
14  from numpy import array
15  from PyQt5 import QtWidgets
16  from PyQt5.QtCore import pyqtSlot
17  from PyQt5.QtGui import QDoubleValidator, QKeySequence
18  from PyQt5.QtWidgets import (QGridLayout, QHBoxLayout, QLabel, QLineEdit,
19                               QPushButton, QShortcut, QSizePolicy, QSpacerItem,
20                               QVBoxLayout)
21
22  from lintrans.gui.dialogs.misc import FixedSizeDialog
23  from lintrans.gui.plots import DefineMatrixVisuallyWidget
24  from lintrans.gui.settings import DisplaySettings
25  from lintrans.gui.validate import MatrixExpressionValidator
26  from lintrans.matrices import MatrixWrapper
27  from lintrans.matrices.utility import is_valid_float, round_float
28  from lintrans.typing_ import MatrixType
29
30  _ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
31
32
33  def get_first_undefined_matrix(wrapper: MatrixWrapper) -> str:
34      """Return the letter of the first undefined matrix in the given wrapper, or ``A`` if all matrices are
    ↪  defined."""
35      defined_matrices = [x for x, _ in wrapper.get_defined_matrices()]
36      for letter in _ALPHABET_NO_I:
37          if letter not in defined_matrices:
38              return letter
39
40      return 'A'
41
42
43  class DefineMatrixDialog(FixedSizeDialog):
44      """An abstract superclass for definitions dialogs.
45
46      .. warning:: This class should never be directly instantiated, only subclassed.
47      """
48
49      def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
```

```
50              """Create the widgets and layout of the dialog.
51
52              .. note:: ``*args`` and ``**kwargs`` are passed to the super constructor (:class:`QDialog`).
53
54              :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
55              """
56              super().__init__(*args, **kwargs)
57
58              self.matrix_wrapper = matrix_wrapper
59              self.setWindowTitle('Define a matrix')
60
61              # === Create the widgets
62
63              self._button_confirm = QPushButton(self)
64              self._button_confirm.setText('Confirm')
65              self._button_confirm.setEnabled(False)
66              self._button_confirm.clicked.connect(self._confirm_matrix)
67              self._button_confirm.setToolTip('Confirm this as the new matrix<br><b>(Ctrl + Enter)</b>')
68              QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self._button_confirm.click)
69
70              button_cancel = QPushButton(self)
71              button_cancel.setText('Cancel')
72              button_cancel.clicked.connect(self.reject)
73              button_cancel.setToolTip('Cancel this definition<br><b>(Escape)</b>')
74
75              label_equals = QLabel(self)
76              label_equals.setText('=')
77
78              self._combobox_letter = QtWidgets.QComboBox(self)
79
80              for letter in _ALPHABET_NO_I:
81                  self._combobox_letter.addItem(letter)
82
83              self._combobox_letter.activated.connect(self._load_matrix)
84              self._combobox_letter.setCurrentText(get_first_undefined_matrix(self.matrix_wrapper))
85
86              # === Arrange the widgets
87
88              self.setContentsMargins(10, 10, 10, 10)
89
90              self._hlay_buttons = QHBoxLayout()
91              self._hlay_buttons.setSpacing(20)
92              self._hlay_buttons.addItem(QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum))
93              self._hlay_buttons.addWidget(button_cancel)
94              self._hlay_buttons.addWidget(self._button_confirm)
95
96              self._hlay_definition = QHBoxLayout()
97              self._hlay_definition.setSpacing(20)
98              self._hlay_definition.addWidget(self._combobox_letter)
99              self._hlay_definition.addWidget(label_equals)
100
101             # All subclasses have to manually add the hlay layouts to _vlay_all
102             # This is because the subclasses add their own widgets and if we add
103             # the layout here, then these new widgets won't be included
104             self._vlay_all = QVBoxLayout()
105             self._vlay_all.setSpacing(20)
106
107             self.setLayout(self._vlay_all)
108
109         @property
110         def _selected_letter(self) -> str:
111             """Return the letter currently selected in the combo box."""
112             return str(self._combobox_letter.currentText())
113
114         @abc.abstractmethod
115         @pyqtSlot()
116         def _update_confirm_button(self) -> None:
117             """Enable the confirm button if it should be enabled, else, disable it."""
118
119         @pyqtSlot(int)
120         def _load_matrix(self, index: int) -> None:
121             """Load the selected matrix into the dialog.
122
```

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

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

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

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


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

    def __init__(
            self,
            *args,
            matrix_wrapper: MatrixWrapper,
            display_settings: DisplaySettings,
            polygon_points: List[Tuple[float, float]],
            **kwargs
    ):
        """Create the widgets and layout of the dialog.

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

        self.setMinimumSize(700, 550)

        # === Create the widgets

        self._plot = DefineMatrixVisuallyWidget(
            self,
            display_settings=display_settings,
            polygon_points=polygon_points
        )

        # === Arrange the widgets

        self._hlay_definition.addWidget(self._plot)
        self._hlay_definition.setStretchFactor(self._plot, 1)

        self._vlay_all.addLayout(self._hlay_definition)
        self._vlay_all.addLayout(self._hlay_buttons)

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

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

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

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

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

```

```
195             if matrix is None:
196                 matrix = self.matrix_wrapper['I']
197
198             self._plot.plot_matrix(matrix)
199             self._plot.update()
200
201         @pyqtSlot()
202         def _confirm_matrix(self) -> None:
203             """Confirm the matrix that's been defined visually."""
204             matrix: MatrixType = array([
205                 [self._plot.point_i[0], self._plot.point_j[0]],
206                 [self._plot.point_i[1], self._plot.point_j[1]]
207             ])
208
209             self.matrix_wrapper[self._selected_letter] = matrix
210             self.accept()
211
212
213     class DefineNumericallyDialog(DefineMatrixDialog):
214         """The dialog class that allows the user to define a new matrix numerically."""
215
216         def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
217             """Create the widgets and layout of the dialog.
218
219             :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
220             """
221             super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
222
223             # === Create the widgets
224
225             # tl = top left, br = bottom right, etc.
226             self._element_tl = QLineEdit(self)
227             self._element_tl.textChanged.connect(self._update_confirm_button)
228             self._element_tl.setValidator(QDoubleValidator())
229
230             self._element_tr = QLineEdit(self)
231             self._element_tr.textChanged.connect(self._update_confirm_button)
232             self._element_tr.setValidator(QDoubleValidator())
233
234             self._element_bl = QLineEdit(self)
235             self._element_bl.textChanged.connect(self._update_confirm_button)
236             self._element_bl.setValidator(QDoubleValidator())
237
238             self._element_br = QLineEdit(self)
239             self._element_br.textChanged.connect(self._update_confirm_button)
240             self._element_br.setValidator(QDoubleValidator())
241
242             self._matrix_elements = (self._element_tl, self._element_tr, self._element_bl, self._element_br)
243
244             font_parens = self.font()
245             font_parens.setPointSize(int(font_parens.pointSize() * 5))
246             font_parens.setWeight(int(font_parens.weight() / 5))
247
248             label_paren_left = QLabel(self)
249             label_paren_left.setText('(')
250             label_paren_left.setFont(font_parens)
251
252             label_paren_right = QLabel(self)
253             label_paren_right.setText(')')
254             label_paren_right.setFont(font_parens)
255
256             # === Arrange the widgets
257
258             grid_matrix = QGridLayout()
259             grid_matrix.setSpacing(20)
260             grid_matrix.addWidget(label_paren_left, 0, 0, -1, 1)
261             grid_matrix.addWidget(self._element_tl, 0, 1)
262             grid_matrix.addWidget(self._element_tr, 0, 2)
263             grid_matrix.addWidget(self._element_bl, 1, 1)
264             grid_matrix.addWidget(self._element_br, 1, 2)
265             grid_matrix.addWidget(label_paren_right, 0, 3, -1, 1)
266
267             self._hlay_definition.addLayout(grid_matrix)
```

```
268
269             self._vlay_all.addLayout(self._hlay_definition)
270             self._vlay_all.addLayout(self._hlay_buttons)
271
272             # We load the default matrix A into the boxes
273             self._load_matrix(0)
274
275             self._element_tl.setFocus()
276
277         @pyqtSlot()
278         def _update_confirm_button(self) -> None:
279             """Enable the confirm button if there are valid floats in every box."""
280             for elem in self._matrix_elements:
281                 if not is_valid_float(elem.text()):
282                     # If they're not all numbers, then we can't confirm it
283                     self._button_confirm.setEnabled(False)
284                     return
285
286             # If we didn't find anything invalid
287             self._button_confirm.setEnabled(True)
288
289         @pyqtSlot(int)
290         def _load_matrix(self, index: int) -> None:
291             """If the selected matrix is defined, load its values into the boxes."""
292             matrix = self.matrix_wrapper[self._selected_letter]
293
294             if matrix is None:
295                 for elem in self._matrix_elements:
296                     elem.setText('')
297
298             else:
299                 self._element_tl.setText(round_float(matrix[0][0]))
300                 self._element_tr.setText(round_float(matrix[0][1]))
301                 self._element_bl.setText(round_float(matrix[1][0]))
302                 self._element_br.setText(round_float(matrix[1][1]))
303
304             self._update_confirm_button()
305
306         @pyqtSlot()
307         def _confirm_matrix(self) -> None:
308             """Confirm the matrix in the boxes and assign it to the name in the combo box."""
309             matrix: MatrixType = array([
310                 [float(self._element_tl.text()), float(self._element_tr.text())],
311                 [float(self._element_bl.text()), float(self._element_br.text())]
312             ])
313
314             self.matrix_wrapper[self._selected_letter] = matrix
315             self.accept()
316
317
318     class DefineAsExpressionDialog(DefineMatrixDialog):
319         """The dialog class that allows the user to define a matrix as an expression of other matrices."""
320
321         def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
322             """Create the widgets and layout of the dialog.
323
324             :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
325             """
326             super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
327
328             self.setMinimumWidth(450)
329
330             # === Create the widgets
331
332             self._lineedit_expression_box = QLineEdit(self)
333             self._lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
334             self._lineedit_expression_box.textChanged.connect(self._update_confirm_button)
335             self._lineedit_expression_box.setValidator(MatrixExpressionValidator())
336
337             # === Arrange the widgets
338
339             self._hlay_definition.addWidget(self._lineedit_expression_box)
340
```

```
341             self._vlay_all.addLayout(self._hlay_definition)
342             self._vlay_all.addLayout(self._hlay_buttons)
343
344             # Load the matrix if it's defined as an expression
345             self._load_matrix(0)
346
347             self._lineedit_expression_box.setFocus()
348
349         @pyqtSlot()
350         def _update_confirm_button(self) -> None:
351             """Enable the confirm button if the matrix expression is valid in the wrapper."""
352             text = self._lineedit_expression_box.text()
353             valid_expression = self.matrix_wrapper.is_valid_expression(text)
354
355             self._button_confirm.setEnabled(
356                 valid_expression
357                 and self._selected_letter not in text
358                 and self._selected_letter not in self.matrix_wrapper.get_expression_dependencies(text)
359             )
360
361         @pyqtSlot(int)
362         def _load_matrix(self, index: int) -> None:
363             """If the selected matrix is defined an expression, load that expression into the box."""
364             if (expr := self.matrix_wrapper.get_expression(self._selected_letter)) is not None:
365                 self._lineedit_expression_box.setText(expr)
366             else:
367                 self._lineedit_expression_box.setText('')
368
369         @pyqtSlot()
370         def _confirm_matrix(self) -> None:
371             """Evaluate the matrix expression and assign its value to the name in the combo box."""
372             self.matrix_wrapper[self._selected_letter] = self._lineedit_expression_box.text()
373             self.accept()
```

A.19 `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 + ')?'
```

```
        real_number = f'({integer_no_zero}(\\.{digits})?|0\\.{digits})'

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

        return re.compile(expression)


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


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

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

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

        while True:
            char = expression[pointer]

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

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

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

            if paren_depth > 0:
                string += char

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

            pointer += 1

            if pointer >= len(expression):
                break

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

        return sub_expressions


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

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

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

```
        match = _naive_expression_pattern.match(expression)

        if match is None:
            return False

        if re.search(r'\^-?\d*\.\d+', expression) is not None:
            return False

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

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

        if 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 = re.sub(r'\s', '', expression)

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

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

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

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

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

            self._expression = expression
```

```
180             self._pointer: int = 0
181
182             self._current_token = MatrixToken()
183             self._current_group: List[Tuple[str, str, str]] = []
184
185             self._final_list: MatrixParseList = []
186
187         def __repr__(self) -> str:
188             """Return a simple repr containing the expression."""
189             return f'{self.__class__.__module__}.{self.__class__.__name__}("{self._expression}")'
190
191         @property
192         def _char(self) -> str:
193             """Return the character pointed to by the pointer."""
194             return self._expression[self._pointer]
195
196         def parse(self) -> MatrixParseList:
197             """Fully parse the instance's matrix expression and return the :attr:`~lintrans.typing_.MatrixParseList`.
198
199             This method uses all the private methods of this class to parse the
200             expression in parts. All private methods mutate the instance variables.
201
202             :returns: The parsed expression
203             :rtype: :attr:`~lintrans.typing_.MatrixParseList`
204             """
205             self._parse_multiplication_group()
206
207             while self._pointer < len(self._expression):
208                 if self._expression[self._pointer] != '+':
209                     raise MatrixParseError('Expected "+" between multiplication groups')
210
211                 self._pointer += 1
212                 self._parse_multiplication_group()
213
214             return self._final_list
215
216         def _parse_multiplication_group(self) -> None:
217             """Parse a group of matrices to be multiplied together.
218
219             This method just parses matrices until we get to a ``+``.
220             """
221             # This loop continues to parse matrices until we fail to do so
222             while self._parse_matrix():
223                 # Once we get to the end of the multiplication group, we add it the final list and reset the group list
224                 if self._pointer >= len(self._expression) or self._char == '+':
225                     self._final_list.append(self._current_group)
226                     self._current_group = []
227                     self._pointer += 1
228
229         def _parse_matrix(self) -> bool:
230             """Parse a full matrix using :meth:`_parse_matrix_part`.
231
232             This method will parse an optional multiplier, an identifier, and an optional exponent. If we
233             do this successfully, we return True. If we fail to parse a matrix (maybe we've reached the
234             end of the current multiplication group and the next char is ``+``), then we return False.
235
236             :returns bool: Success or failure
237             """
238             self._current_token = MatrixToken()
239
240             while self._parse_matrix_part():
241                 pass  # The actual execution is taken care of in the loop condition
242
243             if self._current_token.identifier == '':
244                 return False
245
246             self._current_group.append(self._current_token.tuple)
247             return True
248
249         def _parse_matrix_part(self) -> bool:
250             """Parse part of a matrix (multiplier, identifier, or exponent).
251
252             Which part of the matrix we parse is dependent on the current value of the pointer and the expression.
```

```
253             This method will parse whichever part of matrix token that it can. If it can't parse a part of a matrix,
254             or it's reached the next matrix, then we just return False. If we succeeded to parse a matrix part, then
255             we return True.
256
257             :returns bool: Success or failure
258             :raises MatrixParseError: If we fail to parse this part of the matrix
259             """
260             if self._pointer >= len(self._expression):
261                 return False
262
263             if self._char.isdigit() or self._char == '-':
264                 if self._current_token.multiplier != '' \
265                         or (self._current_token.multiplier == '' and self._current_token.identifier != ''):
266                     return False
267
268                 self._parse_multiplier()
269
270             elif self._char.isalpha() and self._char.isupper():
271                 if self._current_token.identifier != '':
272                     return False
273
274                 self._current_token.identifier = self._char
275                 self._pointer += 1
276
277             elif self._char == 'r':
278                 if self._current_token.identifier != '':
279                     return False
280
281                 self._parse_rot_identifier()
282
283             elif self._char == '(':
284                 if self._current_token.identifier != '':
285                     return False
286
287                 self._parse_sub_expression()
288
289             elif self._char == '^':
290                 if self._current_token.exponent != '':
291                     return False
292
293                 self._parse_exponent()
294
295             elif self._char == '+':
296                 return False
297
298             else:
299                 raise MatrixParseError(f'Unrecognised character "{self._char}" in matrix expression')
300
301             return True
302
303         def _parse_multiplier(self) -> None:
304             """Parse a multiplier from the expression and pointer.
305
306             This method just parses a numerical multiplier, which can include
307             zero or one ``.`` character and optionally a ``-`` at the start.
308
309             :raises MatrixParseError: If we fail to parse this part of the matrix
310             """
311             multiplier = ''
312
313             while self._char.isdigit() or self._char in ('.', '-'):
314                 multiplier += self._char
315                 self._pointer += 1
316
317             try:
318                 float(multiplier)
319             except ValueError as e:
320                 raise MatrixParseError(f'Invalid multiplier "{multiplier}"') from e
321
322             self._current_token.multiplier = multiplier
323
324         def _parse_rot_identifier(self) -> None:
325             """Parse a ``rot()``-style identifier from the expression and pointer.
```

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

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

            self._current_token.identifier = match.group(0)
            self._pointer += len(match.group(0))
        else:
            raise MatrixParseError(
                f'Invalid rot-identifier "{self._expression[self._pointer : self._pointer + 15]}..."'
            )

    def _parse_sub_expression(self) -> None:
        """Parse a parenthesized sub-expression as the identifier.

        This method will also validate the expression in the parentheses.

        :raises MatrixParseError: If we fail to parse this part of the matrix
        """
        if self._char != '(':
            raise MatrixParseError('Sub-expression must start with "("')

        self._pointer += 1
        paren_depth = 1
        identifier = ''

        while paren_depth > 0:
            if self._char == '(':
                paren_depth += 1
            elif self._char == ')':
                paren_depth -= 1

            if paren_depth == 0:
                self._pointer += 1
                break

            identifier += self._char
            self._pointer += 1

        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]}..."'
            )
```

```python

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.20 `matrices/wrapper.py`

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

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

"""This module contains the main :class:`MatrixWrapper` class and a function to create a matrix from an angle."""

from __future__ import annotations

import re
from copy import copy
from functools import reduce
from operator import add, matmul
from typing import Any, Dict, List, Optional, 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

_ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'


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

    .. note::
       When defining a custom matrix, its name must be a capital letter and cannot be ``I``.

    The contained matrices can be accessed and assigned to using square bracket notation.

    :Example:

    >>> wrapper = MatrixWrapper()
    >>> wrapper['I']
    array([[1., 0.],
           [0., 1.]])
    >>> wrapper['M']  # Returns None
    >>> wrapper['M'] = np.array([[1, 2], [3, 4]])
    >>> wrapper['M']
    array([[1., 2.],
           [3., 4.]])
    """

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

    def __repr__(self) -> str:
        """Return a nice string repr of the :class:`MatrixWrapper` for debugging."""
        defined_matrices = ''.join([k for k, v in self._matrices.items() if v is not None])
        return f'<{self.__class__.__module__}.{self.__class__.__name__} object with ' \
               f"{len(defined_matrices)} defined matrices: '{defined_matrices}'>"

    def __eq__(self, other: Any) -> bool:
        """Check for equality in wrappers by comparing dictionaries.

        :param Any other: The object to compare this wrapper to
        """
        if not isinstance(other, self.__class__):
            return NotImplemented

        # We loop over every matrix and check if every value is equal in each
        for name in self._matrices:
            s_matrix = self[name]
            o_matrix = other[name]

            if s_matrix is None and o_matrix is None:
                continue

            elif (s_matrix is None and o_matrix is not None) or \
                 (s_matrix is not None and o_matrix is None):
                return False

            # This is mainly to satisfy mypy, because we know these must be matrices
            elif not is_matrix_type(s_matrix) or not is_matrix_type(o_matrix):
                return False

            # Now we know they're both NumPy arrays
```

```
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 name.
107
108           If it is a simple name, it will just be fetched from the dictionary. If the name is ``rot(x)``, with
109           a given angle in degrees, then we return a new matrix representing a rotation by that angle.
110
111           .. note::
112              If the named matrix is defined as an expression, then this method will return its evaluation.
113              If you want the expression itself, use :meth:`get_expression`.
114
115           :param str name: The name of the matrix to get
116           :returns Optional[MatrixType]: The value of the matrix (could be None)
117
118           :raises NameError: If there is no matrix with the given name
119           """
120           # Return a new rotation matrix
121           if (match := re.match(r'^rot\((-?\d*\.?\d*)\)$', name)) is not None:
122               return create_rotation_matrix(float(match.group(1)))
123
124           if name not in self._matrices:
125               if validate_matrix_expression(name):
126                   return self.evaluate_expression(name)
127
128               raise NameError(f'Unrecognised matrix name "{name}"')
129
130           # We copy the matrix before we return it so the user can't accidentally mutate the matrix
131           matrix = copy(self._matrices[name])
132
133           if isinstance(matrix, str):
134               return self.evaluate_expression(matrix)
135
136           return matrix
137
138       def __setitem__(self, name: str, new_matrix: Optional[Union[MatrixType, str]]) -> None:
139           """Set the value of matrix ``name`` with the new_matrix.
140
141           The new matrix may be a simple 2x2 NumPy array, or it could be a string, representing an
142           expression in terms of other, previously defined matrices.
143
144           :param str name: The name of the matrix to set the value of
145           :param Optional[Union[MatrixType, str]] new_matrix: The value of the new matrix (could be None)
146
147           :raises NameError: If the name isn't a legal matrix name
148           :raises TypeError: If the matrix isn't a valid 2x2 NumPy array or expression in terms of other defined
    ↪  matrices
149           :raises ValueError: If you attempt to define a matrix in terms of itself
150           """
151           if not (name in self._matrices and name != 'I'):
152               raise NameError('Matrix name is illegal')
153
154           if new_matrix is None:
155               self._matrices[name] = None
156               return
157
158           if isinstance(new_matrix, str):
159               if self.is_valid_expression(new_matrix):
160                   if name not in new_matrix and \
161                           name not in self.get_expression_dependencies(new_matrix):
162                       self._matrices[name] = new_matrix
163                       return
164                   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:
        """Check if the given expression is valid, using the context of the wrapper.

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

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

        :raises LinAlgError: If a matrix is defined in terms of the inverse of a singular matrix
```

```
238             """
239             # Get rid of the transposes to check all capital letters
240             new_expression = expression.replace('^T', '').replace('^{T}', '')
241 
242             # Make sure all the referenced matrices are defined
243             for matrix in [x for x in new_expression if re.match('[A-Z]', x)]:
244                 if self[matrix] is None:
245                     return False
246 
247                 if (expr := self.get_expression(matrix)) is not None:
248                     if not self.is_valid_expression(expr):
249                         return False
250 
251             return validate_matrix_expression(expression)
252 
253         def evaluate_expression(self, expression: str) -> MatrixType:
254             """Evaluate a given expression and return the matrix evaluation.
255 
256             :param str expression: The expression to be parsed
257             :returns MatrixType: The matrix result of the expression
258 
259             :raises ValueError: If the expression is invalid
260             """
261             if not self.is_valid_expression(expression):
262                 raise ValueError('The expression is invalid')
263 
264             parsed_result = parse_matrix_expression(expression)
265             final_groups: List[List[MatrixType]] = []
266 
267             for group in parsed_result:
268                 f_group: List[MatrixType] = []
269 
270                 for multiplier, identifier, index in group:
271                     if index == 'T':
272                         m = self[identifier]
273 
274                         # This assertion is just so mypy doesn't complain
275                         # We know this won't be None, because we know that this matrix is defined in this wrapper
276                         assert m is not None
277                         matrix_value = m.T
278 
279                     else:
280                         # Again, this assertion is just for mypy
281                         # We know this will be a matrix, but since upgrading from NumPy 1.21 to 1.23
282                         # (to fix a bug with GH Actions on Windows), mypy complains about matrix_power()
283                         base_matrix = self[identifier]
284                         assert is_matrix_type(base_matrix)
285 
286                         matrix_value = np.linalg.matrix_power(base_matrix, 1 if index == '' else int(index))
287 
288                     matrix_value *= 1 if multiplier == '' else float(multiplier)
289                     f_group.append(matrix_value)
290 
291                 final_groups.append(f_group)
292 
293             return reduce(add, [reduce(matmul, group) for group in final_groups])
294 
295         def get_defined_matrices(self) -> List[Tuple[str, Union[MatrixType, str]]]:
296             """Return a list of tuples containing the name and value of all defined matrices in the wrapper.
297 
298             :returns: A list of tuples where the first element is the name, and the second element is the value
299             :rtype: List[Tuple[str, Union[MatrixType, str]]]
300             """
301             matrices = []
302 
303             for name, value in self._matrices.items():
304                 if value is not None:
305                     matrices.append((name, value))
306 
307             return matrices
308 
309         def undefine_matrix(self, name: str) -> Set[str]:
310             """Safely undefine the given matrix by also undefining any matrices that depend on it."""
```

```
311             if not (name in self._matrices and name != 'I'):
312                 raise NameError('Matrix name is illegal')
313 
314             # This maps each matrix to all the matrices that depend on it
315             dependents_map = {
316                 x: set(y for y in _ALPHABET_NO_I if x in self.get_matrix_dependencies(y))
317                 for x in _ALPHABET_NO_I
318             }
319 
320             s: Set[str] = set(name)
321             self[name] = None
322             for x in dependents_map[name]:
323                 s.update(self.undefine_matrix(x))
324 
325             return s
```

A.21 matrices/__init__.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3 
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6 
7   """This package 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.22 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
44      def rotate_coord(x: float, y: float, angle: float, *, degrees: bool = False) -> Tuple[float, float]:
45          """Rotate a rectilinear coordinate by the given angle."""
46          if degrees:
47              angle = np.radians(angle)
48
49          r, theta = polar_coords(x, y, degrees=degrees)
50          theta = (theta + angle) % (2 * np.pi)
51
52          return rect_coords(r, theta, degrees=degrees)
53
54
55      def create_rotation_matrix(angle: float, *, degrees: bool = True) -> MatrixType:
56          """Create a matrix representing a rotation (anticlockwise) by the given angle.
57
58          :Example:
59
60          >>> create_rotation_matrix(30)
61          array([[ 0.8660254, -0.5      ],
62                 [ 0.5      ,  0.8660254]])
63          >>> create_rotation_matrix(45)
64          array([[ 0.70710678, -0.70710678],
65                 [ 0.70710678,  0.70710678]])
66          >>> create_rotation_matrix(np.pi / 3, degrees=False)
67          array([[ 0.5      , -0.8660254],
68                 [ 0.8660254,  0.5      ]])
69
70          :param float angle: The angle to rotate anticlockwise by
71          :param bool degrees: Whether to interpret the angle as degrees (True) or radians (False)
72          :returns MatrixType: The resultant matrix
73          """
74          rad = np.deg2rad(angle % 360) if degrees else angle % (2 * np.pi)
75          return np.array([
76              [np.cos(rad), -1 * np.sin(rad)],
77              [np.sin(rad), np.cos(rad)]
78          ])
79
80
81      def is_valid_float(string: str) -> bool:
82          """Check if the string is a valid float (or anything that can be cast to a float, such as an int).
83
84          This function simply checks that ``float(string)`` doesn't raise an error.
85
86          .. note:: An empty string is not a valid float, so will return False.
87
88          :param str string: The string to check
89          :returns bool: Whether the string is a valid float
90          """
91          try:
92              float(string)
93              return True
94          except ValueError:
95              return False
96
97
98      def round_float(num: float, precision: int = 5) -> str:
99          """Round a floating point number to a given number of decimal places for pretty printing.
100
101         :param float num: The number to round
102         :param int precision: The number of decimal places to round to
103         :returns str: The rounded number for pretty printing
104         """
105         # Round to ``precision`` number of decimal places
106         string = str(round(num, precision))
107
```

```
        # Cut off the potential final zero
        if string.endswith('.0'):
            return string[:-2]

        elif 'e' in string:  # Scientific notation
            split = string.split('e')
            # The leading 0 only happens when the exponent is negative, so we know there'll be a minus sign
            return split[0] + 'e-' + split[1][1:].lstrip('0')

        else:
            return string
```

A.23 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
from numpy import ndarray

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

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

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

VectorType: TypeAlias = 'NDArray[(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)
```

B Testing code

B.1 `conftest.py`

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

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

"""A simple ``conftest.py`` containing some re-usable fixtures and functions."""

import os
from typing import List, Type, TypeVar

import numpy as np
import pytest
from _pytest.config import Config
from _pytest.python import Function
from PyQt5.QtWidgets import QApplication, QWidget
from pytestqt.qtbot import QtBot

from lintrans.gui.main_window import LintransMainWindow
from lintrans.matrices import MatrixWrapper

T = TypeVar('T', bound=QWidget)


def pytest_collection_modifyitems(config: Config, items: List[Function]) -> None:
    """Modify the collected tests so that we only run the GUI tests on Linux (because they need an X server).

    This function is called automatically during the pytest startup. See
    https://docs.pytest.org/en/latest/example/simple.html#control-skipping-of-tests-according-to-command-line-option
    for details.
    """
    skip_gui = pytest.mark.skip(reason='need X server (Linux only) to run GUI tests')
    for item in items:
        if 'gui' in item.location[0] and hasattr(os, 'uname') and os.uname().sysname != 'Linux':
            item.add_marker(skip_gui)


# === Backend stuff

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

    root_two_over_two = np.sqrt(2) / 2

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

    return wrapper


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


@pytest.fixture
def new_wrapper() -> MatrixWrapper:
```

```
68          """Return a new MatrixWrapper with no initialized values."""
69          return MatrixWrapper()
70
71
72      # === GUI stuff
73
74      def is_widget_class_open(widget_class: Type[QWidget]) -> bool:
75          """Test if a widget with the given class is currently open."""
76          return widget_class in [x.__class__ for x in QApplication.topLevelWidgets()]
77
78
79      @pytest.fixture
80      def window(qtbot: QtBot) -> LintransMainWindow:
81          """Return an instance of :class:`LintransMainWindow`."""
82          window = LintransMainWindow()
83          qtbot.addWidget(window)
84          return window
85
86
87      def get_open_widget(widget_class: Type[T]) -> T:
88          """Get the open instance of the given :class:`QWidget` subclass.
89
90          This method assumes that there is exactly 1 widget of the given
91          class and will raise ``ValueError`` if there's not.
92
93          :raises ValueError: If there is not exactly one widget of the given class
94          """
95          widgets = [
96              x for x in QApplication.topLevelWidgets()
97              if isinstance(x, widget_class)
98          ]
99
100         if len(widgets) != 1:
101             raise ValueError(f'Expected 1 widget of type {widget_class} but found {len(widgets)}')
102
103         return widgets[0]
```

B.2 gui/test_define_dialogs.py

```
1       # lintrans - The linear transformation visualizer
2       # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4       # This program is licensed under GNU GPLv3, available here:
5       # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7       """Test the :class:`DefineDialog` boxes in :class:`LintransMainWindow`."""
8
9       import numpy as np
10      from conftest import get_open_widget, is_widget_class_open
11      from PyQt5.QtCore import Qt
12      from pytestqt.qtbot import QtBot
13
14      from lintrans.gui.dialogs import (DefineAsExpressionDialog,
15                                        DefineNumericallyDialog,
16                                        DefineVisuallyDialog)
17      from lintrans.gui.main_window import LintransMainWindow
18
19      ALPHABET_NO_I = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
20
21
22      def test_define_visually_dialog_opens(qtbot: QtBot, window: LintransMainWindow) -> None:
23          """Test that the :class:`DefineVisuallyDialog` opens."""
24          qtbot.mouseClick(window._button_define_visually, Qt.LeftButton)
25          assert is_widget_class_open(DefineVisuallyDialog)
26          qtbot.addWidget(get_open_widget(DefineVisuallyDialog))
27
28
29      def test_define_numerically_dialog_opens(qtbot: QtBot, window: LintransMainWindow) -> None:
30          """Test that the :class:`DefineNumericallyDialog` opens."""
31          qtbot.mouseClick(window._button_define_numerically, Qt.LeftButton)
```

```
    assert is_widget_class_open(DefineNumericallyDialog)
    qtbot.addWidget(get_open_widget(DefineNumericallyDialog))


def test_define_as_expression_dialog_opens(qtbot: QtBot, window: LintransMainWindow) -> None:
    """Test that the :class:`DefineAsAnExpressionDialog` opens."""
    qtbot.mouseClick(window._button_define_as_expression, Qt.LeftButton)
    assert is_widget_class_open(DefineAsExpressionDialog)
    qtbot.addWidget(get_open_widget(DefineAsExpressionDialog))


def test_define_numerically_dialog_works(qtbot: QtBot, window: LintransMainWindow) -> None:
    """Test that matrices can be defined numerically."""
    qtbot.mouseClick(window._button_define_numerically, Qt.LeftButton)
    dialog = get_open_widget(DefineNumericallyDialog)
    qtbot.addWidget(dialog)

    qtbot.keyClicks(dialog._element_tl, '-1')
    qtbot.keyClicks(dialog._element_tr, '3')
    qtbot.keyClicks(dialog._element_bl, '2')
    qtbot.keyClicks(dialog._element_br, '-0.5')

    qtbot.mouseClick(dialog._button_confirm, Qt.LeftButton)

    assert (window._matrix_wrapper['A'] == np.array([
        [-1, 3],
        [2, -0.5]
    ])).all()


def test_define_as_expression_dialog_works(qtbot: QtBot, window: LintransMainWindow) -> None:
    """Test that matrices can be defined as expressions."""
    qtbot.mouseClick(window._button_define_as_expression, Qt.LeftButton)
    dialog = get_open_widget(DefineAsExpressionDialog)
    qtbot.addWidget(dialog)

    qtbot.keyClicks(dialog._lineedit_expression_box, '(rot(45)^{2}3I)^Trot(210)^-1')
    qtbot.mouseClick(dialog._button_confirm, Qt.LeftButton)

    assert window._matrix_wrapper.get_expression('A') == '(rot(45)^{2}3I)^Trot(210)^-1'
    assert (
        window._matrix_wrapper['A'] ==
        window._matrix_wrapper.evaluate_expression('(rot(45)^{2}3I)^Trot(210)^-1')
    ).all()
```

B.3 gui/test_other_dialogs.py

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

"""Test that the non-defintion dialogs work as expected."""

from typing import Type

import pytest
from conftest import get_open_widget, is_widget_class_open
from PyQt5.QtCore import Qt
from PyQt5.QtWidgets import QDialog
from pytestqt.qtbot import QtBot

from lintrans.gui.dialogs import DisplaySettingsDialog, InfoPanelDialog
from lintrans.gui.main_window import LintransMainWindow


@pytest.mark.parametrize(
    'button_attr,dialog_class',
    [
```

```
        ('_button_change_display_settings', DisplaySettingsDialog),
        ('_button_info_panel', InfoPanelDialog),
    ]
)
def test_dialogs_open(
    qtbot: QtBot,
    window: LintransMainWindow,
    button_attr: str,
    dialog_class: Type[QDialog]
) -> None:
    """Make sure the dialog opens properly."""
    qtbot.mouseClick(getattr(window, button_attr), Qt.LeftButton)
    assert is_widget_class_open(dialog_class)
    qtbot.addWidget(get_open_widget(dialog_class))
```

B.4 backend/test_session.py

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

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

"""Test the functionality of saving and loading sessions."""

from pathlib import Path

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.5 backend/matrices/test_parse_and_validate_expression.py

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

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

"""Test the :mod:`matrices.parse` module validation and parsing."""

from typing import List, Tuple
```

```
10
11      import pytest
12
13      from lintrans.matrices.parse import (MatrixParseError, find_sub_expressions,
14                                           get_matrix_identifiers,
15                                           parse_matrix_expression,
16                                           validate_matrix_expression)
17      from lintrans.typing_ import MatrixParseList
18
19      expected_sub_expressions: List[Tuple[str, List[str]]] = [
20          ('2(AB)^-1', ['AB']),
21          ('-3(A+B)^2-C(B^TA)^-1', ['A+B', 'B^TA']),
22          ('rot(45)', []),
23          ('()', []),
24          ('(())', ['()']),
25          ('2.3A^-1(AB)^-1+(BC)^2', ['AB', 'BC']),
26          ('(2.3A^-1(AB)^-1+(BC)^2)', ['2.3A^-1(AB)^-1+(BC)^2']),
27      ]
28
29
30      def test_find_sub_expressions() -> None:
31          """Test the :func:`lintrans.matrices.parse.find_sub_expressions` function."""
32          for inp, output in expected_sub_expressions:
33              assert find_sub_expressions(inp) == output
34
35
36      valid_inputs: List[str] = [
37          'A', 'AB', '3A', '1.2A', '-3.4A', 'A^2', 'A^-1', 'A^{-1}',
38          'A^12', 'A^T', 'A^{5}', 'A^{T}', '4.3A^7', '9.2A^{18}', '0.1A'
39
40          'rot(45)', 'rot(12.5)', '3rot(90)',
41          'rot(135)^3', 'rot(51)^T', 'rot(-34)^-1',
42
43          'A+B', 'A+2B', '4.3A+9B', 'A^2+B^T', '3A^7+0.8B^{16}',
44          'A-B', '3A-4B', '3.2A^3-16.79B^T', '4.752A^{17}-3.32B^{36}',
45          'A-1B', '-A', '-1A', 'A^{2}3.4B', 'A^{-1}2.3B',
46
47          '3A4B', 'A^TB', 'A^{T}B', '4A^6B^3',
48          '2A^{3}4B^5', '4rot(90)^3', 'rot(45)rot(13)',
49          'Arot(90)', 'AB^2', 'A^2B^2', '8.36A^T3.4B^12',
50
51          '3.5A^{4}5.6rot(19.2)^T-B^{-1}4.1C^5',
52
53          '(A)', '(AB)^-1', '2.3(3B^TA)^2', '-3.4(9D^{2}3F^-1)^T+C', '(AB)(C)',
54          '3(rot(34)^-7A)^-1+B', '3A^2B+4A(B+C)^-1D^T-A(C(D+E)B)'
55      ]
56
57      invalid_inputs: List[str] = [
58          '', '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',
59          '^T', '^{12}', '.1A', 'A^{13', 'A^3}', 'A^A', '^2', 'A--B', '--A', '+A', '--1A', 'A--B', 'A--1B',
60          '.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)',
61          '3()^2', '4(your mum)^T', 'rot()', 'rot(10.1.1)', 'rot(--2)',
62
63          'This is 100% a valid matrix expression, I swear'
64      ]
65
66
67      @pytest.mark.parametrize('inputs,output', [(valid_inputs, True), (invalid_inputs, False)])
68      def test_validate_matrix_expression(inputs: List[str], output: bool) -> None:
69          """Test the validate_matrix_expression() function."""
70          for inp in inputs:
71              assert validate_matrix_expression(inp) == output
72
73
74      expressions_and_parsed_expressions: List[Tuple[str, MatrixParseList]] = [
75          # Simple expressions
76          ('A', [[('', 'A', '')]]),
77          ('A^2', [[('', 'A', '2')]]),
78          ('A^{2}', [[('', 'A', '2')]]),
79          ('3A', [[('3', 'A', '')]]),
80          ('1.4A^3', [[('1.4', 'A', '3')]]),
81          ('0.1A', [[('0.1', 'A', '')]]),
82          ('0.1A', [[('0.1', 'A', '')]]),
```

```
        ('A^12', [[('', 'A', '12')]]),
        ('A^234', [[('', 'A', '234')]]),

        # Multiplications
        ('A 0.1B', [[('', 'A', ''), ('0.1', 'B', '')]]),
        ('A^2 3B', [[('', 'A', '23'), ('', 'B', '')]]),
        ('A^{2}3.4B', [[('', 'A', '2'), ('3.4', 'B', '')]]),
        ('4A^{3} 6B^2', [[('4', 'A', '3'), ('6', 'B', '2')]]),
        ('4.2A^{T} 6.1B^-1', [[('4.2', 'A', 'T'), ('6.1', 'B', '-1')]]),
        ('-1.2A^2 rot(45)^2', [[('-1.2', 'A', '2'), ('', 'rot(45)', '2')]]),
        ('3.2A^T 4.5B^{5} 9.6rot(121.3)', [[('3.2', 'A', 'T'), ('4.5', 'B', '5'), ('9.6', 'rot(121.3)', '')]]),
        ('-1.18A^{-2} 0.1B^{2} 9rot(-34.6)^-1', [[('-1.18', 'A', '-2'), ('0.1', 'B', '2'), ('9', 'rot(-34.6)', '-1')]]),

        # Additions
        ('A + B', [[('', 'A', '')], [('', 'B', '')]]),
        ('A + B - C', [[('', 'A', '')], [('', 'B', '')], [('-1', 'C', '')]]),
        ('A^2 + 0.5B', [[('', 'A', '2')], [('0.5', 'B', '')]]),
        ('2A^3 + 8B^T - 3C^-1', [[('2', 'A', '3')], [('8', 'B', 'T')], [('-3', 'C', '-1')]]),
        ('4.9A^2 - 3rot(134.2)^-1 + 7.6B^8', [[('4.9', 'A', '2')], [('-3', 'rot(134.2)', '-1')], [('7.6', 'B', '8')]]),

        # Additions with multiplication
        ('2.14A^{3} 4.5rot(14.5)^-1 + 8B^T - 3C^-1', [[('2.14', 'A', '3'), ('4.5', 'rot(14.5)', '-1')],
                                                      [('8', 'B', 'T')], [('-3', 'C', '-1')]]),
        ('2.14A^{3} 4.5rot(14.5)^-1 + 8.5B^T 5.97C^14 - 3.14D^{-1} 6.7E^T',
         [[('2.14', 'A', '3'), ('4.5', 'rot(14.5)', '-1')], [('8.5', 'B', 'T'), ('5.97', 'C', '14')],
          [('-3.14', 'D', '-1'), ('6.7', 'E', 'T')]]),

        # Parenthesized expressions
        ('(AB)^-1', [[('', 'AB', '-1')]]),
        ('-3(A+B)^2-C(B^TA)^-1', [[('-3', 'A+B', '2')], [('-1', 'C', ''), ('', 'B^{T}A', '-1')]]),
        ('2.3(3B^TA)^2', [[('2.3', '3B^{T}A', '2')]]),
        ('-3.4(9D^{2}3F^-1)^T+C', [[('-3.4', '9D^{2}3F^{-1}', 'T')], [('', 'C', '')]]),
        ('2.39(3.1A^{-1}2.3B(CD)^-1)^T + (AB^T)^-1', [[('2.39', '3.1A^{-1}2.3B(CD)^{-1}', 'T')], [('', 'AB^{T}',
        ↪  '-1')]])
    ]


    def test_parse_matrix_expression() -> None:
        """Test the parse_matrix_expression() function."""
        for expression, parsed_expression in expressions_and_parsed_expressions:
            # Test it with and without whitespace
            assert parse_matrix_expression(expression) == parsed_expression
            assert parse_matrix_expression(expression.replace(' ', '')) == parsed_expression

        for expression in valid_inputs:
            # Assert that it doesn't raise MatrixParseError
            parse_matrix_expression(expression)


    def test_parse_error() -> None:
        """Test that parse_matrix_expression() raises a MatrixParseError."""
        for expression in invalid_inputs:
            with pytest.raises(MatrixParseError):
                parse_matrix_expression(expression)


    def test_get_matrix_identifiers() -> None:
        """Test that matrix identifiers can be properly found."""
        assert get_matrix_identifiers('M^T') == {'M'}
        assert get_matrix_identifiers('ABCDEF') == {'A', 'B', 'C', 'D', 'E', 'F'}
        assert get_matrix_identifiers('AB^{-1}3Crot(45)2A(B^2C^-1)') == {'A', 'B', 'C'}
        assert get_matrix_identifiers('A^{2}3A^-1A^TA') == {'A'}
        assert get_matrix_identifiers('rot(45)(rot(25)rot(20))^2') == set()

        for expression in invalid_inputs:
            with pytest.raises(MatrixParseError):
                get_matrix_identifiers(expression)
```

B.6 backend/matrices/utility/test_float_utility_functions.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 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/utility/test_coord_conversion.py

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

"""Test conversion between polar and rectilinear coordinates in :mod:`lintrans.matrices.utility`."""

from typing import List, Tuple

from numpy import pi, sqrt
from pytest import approx

from lintrans.matrices.utility import polar_coords, rect_coords

expected_coords: List[Tuple[Tuple[float, float], Tuple[float, float]]] = [
    ((0, 0), (0, 0)),
    ((1, 1), (sqrt(2), pi / 4)),
    ((0, 1), (1, pi / 2)),
    ((1, 0), (1, 0)),
```

```
        ((sqrt(2), sqrt(2)), (2, pi / 4)),
        ((-3, 4), (5, 2.214297436)),
        ((4, -3), (5, 5.639684198)),
        ((5, -0.2), (sqrt(626) / 5, 6.24320662)),
        ((-1.3, -10), (10.08414597, 4.583113976)),
        ((23.4, 0), (23.4, 0)),
        ((pi, -pi), (4.442882938, 1.75 * pi))
    ]


    def test_polar_coords() -> None:
        """Test that :func:`lintrans.matrices.utility.polar_coords` works as expected."""
        for rect, polar in expected_coords:
            assert polar_coords(*rect) == approx(polar)


    def test_rect_coords() -> None:
        """Test that :func:`lintrans.matrices.utility.rect_coords` works as expected."""
        for rect, polar in expected_coords:
            assert rect_coords(*polar) == approx(rect)

        assert rect_coords(1, 0) == approx((1, 0))
        assert rect_coords(1, pi) == approx((-1, 0))
        assert rect_coords(1, 2 * pi) == approx((1, 0))
        assert rect_coords(1, 3 * pi) == approx((-1, 0))
        assert rect_coords(1, 4 * pi) == approx((1, 0))
        assert rect_coords(1, 5 * pi) == approx((-1, 0))
        assert rect_coords(1, 6 * pi) == approx((1, 0))
        assert rect_coords(20, 100) == approx(rect_coords(20, 100 % (2 * pi)))
```

B.8 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.9 backend/matrices/matrix_wrapper/test_evaluate_expression.py

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

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

"""Test the MatrixWrapper evaluate_expression() method."""

import numpy as np
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

```

```python

def test_simple_matrix_addition(test_wrapper: MatrixWrapper) -> None:
    """Test simple addition and subtraction of two matrices."""
    # NOTE: We assert that all of these values are not None just to stop mypy complaining
    # These values will never actually be None because they're set in the wrapper() fixture
    # There's probably a better way  do this, because this method is a bit of a bodge, but this works for now
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

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

    assert test_wrapper == get_test_wrapper()


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

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

    assert test_wrapper.evaluate_expression('A2B') == approx(test_wrapper['A'] @ (2 * test_wrapper['B']))
    assert test_wrapper.evaluate_expression('2AB') == approx((2 * test_wrapper['A']) @ test_wrapper['B'])
    assert test_wrapper.evaluate_expression('C3D') == approx(test_wrapper['C'] @ (3 * test_wrapper['D']))
    assert test_wrapper.evaluate_expression('4.2E1.2A') == approx((4.2 * test_wrapper['E']) @ (1.2 *
    ↪  test_wrapper['A']))

    assert test_wrapper == get_test_wrapper()


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

    assert (test_wrapper.evaluate_expression('I') == test_wrapper['I']).all()
    assert (test_wrapper.evaluate_expression('AI') == test_wrapper['A']).all()
    assert (test_wrapper.evaluate_expression('IA') == test_wrapper['A']).all()
    assert (test_wrapper.evaluate_expression('GI') == test_wrapper['G']).all()
    assert (test_wrapper.evaluate_expression('IG') == test_wrapper['G']).all()

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

    assert test_wrapper == get_test_wrapper()


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

```

```
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()
135         assert (test_wrapper.evaluate_expression('D^12') == la.matrix_power(test_wrapper['D'], 12)).all()
136         assert (test_wrapper.evaluate_expression('E^8') == la.matrix_power(test_wrapper['E'], 8)).all()
137         assert (test_wrapper.evaluate_expression('F^{-6}') == la.matrix_power(test_wrapper['F'], -6)).all()
138         assert (test_wrapper.evaluate_expression('G^-2') == la.matrix_power(test_wrapper['G'], -2)).all()
139
140         assert test_wrapper == get_test_wrapper()
141
142
143     def test_matrix_transpose(test_wrapper: MatrixWrapper) -> None:
144         """Test matrix transpositions."""
145         assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
146                test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
147                test_wrapper['G'] is not None
148
149         assert (test_wrapper.evaluate_expression('A^{T}') == test_wrapper['A'].T).all()
150         assert (test_wrapper.evaluate_expression('B^{T}') == test_wrapper['B'].T).all()
151         assert (test_wrapper.evaluate_expression('C^{T}') == test_wrapper['C'].T).all()
152         assert (test_wrapper.evaluate_expression('D^{T}') == test_wrapper['D'].T).all()
153         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()


    def test_complicated_expressions(test_wrapper: MatrixWrapper) -> None:
        """Test evaluation of complicated expressions."""
        assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
               test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
               test_wrapper['G'] is not None

        assert (test_wrapper.evaluate_expression('-3.2A^T 4B^{-1} 6C^{-1} + 8.1D^{2} 3.2E^4') ==
                (-3.2 * test_wrapper['A'].T) @ (4 * la.inv(test_wrapper['B'])) @ (6 * la.inv(test_wrapper['C']))
                + (8.1 * la.matrix_power(test_wrapper['D'], 2)) @ (3.2 * la.matrix_power(test_wrapper['E'], 4))).all()

        assert (test_wrapper.evaluate_expression('53.6D^{2} 3B^T - 4.9F^{2} 2D + A^3 B^-1') ==
                (53.6 * la.matrix_power(test_wrapper['D'], 2)) @ (3 * test_wrapper['B'].T)
                - (4.9 * la.matrix_power(test_wrapper['F'], 2)) @ (2 * test_wrapper['D'])
                + la.matrix_power(test_wrapper['A'], 3) @ la.inv(test_wrapper['B'])).all()

        assert test_wrapper == get_test_wrapper()
```

```python
def test_parenthesized_expressions(test_wrapper: MatrixWrapper) -> None:
    """Test evaluation of parenthesized expressions."""
    assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
           test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
           test_wrapper['G'] is not None

    assert (test_wrapper.evaluate_expression('(A^T)^2') == la.matrix_power(test_wrapper['A'].T, 2)).all()
    assert (test_wrapper.evaluate_expression('(B^T)^3') == la.matrix_power(test_wrapper['B'].T, 3)).all()
    assert (test_wrapper.evaluate_expression('(C^T)^4') == la.matrix_power(test_wrapper['C'].T, 4)).all()
    assert (test_wrapper.evaluate_expression('(D^T)^5') == la.matrix_power(test_wrapper['D'].T, 5)).all()
    assert (test_wrapper.evaluate_expression('(E^T)^6') == la.matrix_power(test_wrapper['E'].T, 6)).all()
    assert (test_wrapper.evaluate_expression('(F^T)^7') == la.matrix_power(test_wrapper['F'].T, 7)).all()
    assert (test_wrapper.evaluate_expression('(G^T)^8') == la.matrix_power(test_wrapper['G'].T, 8)).all()

    assert (test_wrapper.evaluate_expression('(rot(45)^1)^T') == create_rotation_matrix(45).T).all()
    assert (test_wrapper.evaluate_expression('(rot(45)^2)^T') == la.matrix_power(create_rotation_matrix(45),
    ↪  2).T).all()
    assert (test_wrapper.evaluate_expression('(rot(45)^3)^T') == la.matrix_power(create_rotation_matrix(45),
    ↪  3).T).all()
    assert (test_wrapper.evaluate_expression('(rot(45)^4)^T') == la.matrix_power(create_rotation_matrix(45),
    ↪  4).T).all()
    assert (test_wrapper.evaluate_expression('(rot(45)^5)^T') == la.matrix_power(create_rotation_matrix(45),
    ↪  5).T).all()

    assert (test_wrapper.evaluate_expression('D^3(A+6.2F-0.397G^TE)^-2+A') ==
            la.matrix_power(test_wrapper['D'], 3) @ la.matrix_power(
                test_wrapper['A'] + 6.2 * test_wrapper['F'] - 0.397 * test_wrapper['G'].T @ test_wrapper['E'],
                -2
            ) + test_wrapper['A']).all()

    assert (test_wrapper.evaluate_expression('-1.2F^{3}4.9D^T(A^2(B+3E^TF)^-1)^2') ==
            -1.2 * la.matrix_power(test_wrapper['F'], 3) @ (4.9 * test_wrapper['D'].T) @
            la.matrix_power(
                la.matrix_power(test_wrapper['A'], 2) @ la.matrix_power(
                    test_wrapper['B'] + 3 * test_wrapper['E'].T @ test_wrapper['F'],
                    -1
                ),
                2
            )).all()


def test_value_errors(test_wrapper: MatrixWrapper) -> None:
    """Test that evaluate_expression() raises a ValueError for any malformed input."""
    invalid_expressions = ['', '+', '-', 'This is not a valid expression', '3+4',
                           'A+2', 'A^', '^2', 'A^-', 'At', 'A^t', '3^2']

    for expression in invalid_expressions:
        with pytest.raises(ValueError):
            test_wrapper.evaluate_expression(expression)


def test_linalgerror() -> None:
    """Test that certain expressions raise np.linalg.LinAlgError."""
    matrix_a: MatrixType = np.array([
        [0, 0],
        [0, 0]
    ])

    matrix_b: MatrixType = np.array([
        [1, 2],
        [1, 2]
    ])

    wrapper = MatrixWrapper()
    wrapper['A'] = matrix_a
    wrapper['B'] = matrix_b

    assert (wrapper.evaluate_expression('A') == matrix_a).all()
    assert (wrapper.evaluate_expression('B') == matrix_b).all()

    with pytest.raises(np.linalg.LinAlgError):
        wrapper.evaluate_expression('A^-1')

```

```
296         with pytest.raises(np.linalg.LinAlgError):
297             wrapper.evaluate_expression('B^-1')
298 
299         assert (wrapper['A'] == matrix_a).all()
300         assert (wrapper['B'] == matrix_b).all()
```

B.10 backend/matrices/matrix_wrapper/test_setting_and_getting.py

```
1   # lintrans - The linear transformation visualizer
2   # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3 
4   # This program is licensed under GNU GPLv3, available here:
5   # <https://www.gnu.org/licenses/gpl-3.0.html>
6 
7   """Test the MatrixWrapper __setitem__() and __getitem__() methods."""
8 
9   from typing import Any, Dict, List
10 
11  import numpy as np
12  import pytest
13  from numpy import linalg as la
14 
15  from lintrans.matrices import MatrixWrapper
16  from lintrans.typing_ import MatrixType
17 
18  valid_matrix_names = 'ABCDEFGHJKLMNOPQRSTUVWXYZ'
19  invalid_matrix_names = ['bad name', '123456', 'Th15 Is an 1nV@l1D n@m3', 'abc', 'a']
20 
21  test_matrix: MatrixType = np.array([[1, 2], [4, 3]])
22 
23 
24  def test_basic_get_matrix(new_wrapper: MatrixWrapper) -> None:
25      """Test MatrixWrapper().__getitem__()."""
26      for name in valid_matrix_names:
27          assert new_wrapper[name] is None
28 
29      assert (new_wrapper['I'] == np.array([[1, 0], [0, 1]])).all()
30 
31 
32  def test_get_name_error(new_wrapper: MatrixWrapper) -> None:
33      """Test that MatrixWrapper().__getitem__() raises a NameError if called with an invalid name."""
34      for name in invalid_matrix_names:
35          with pytest.raises(NameError):
36              _ = new_wrapper[name]
37 
38 
39  def test_basic_set_matrix(new_wrapper: MatrixWrapper) -> None:
40      """Test MatrixWrapper().__setitem__()."""
41      for name in valid_matrix_names:
42          new_wrapper[name] = test_matrix
43          assert (new_wrapper[name] == test_matrix).all()
44 
45          new_wrapper[name] = None
46          assert new_wrapper[name] is None
47 
48 
49  def test_set_expression(test_wrapper: MatrixWrapper) -> None:
50      """Test that MatrixWrapper.__setitem__() can accept a valid expression."""
51      test_wrapper['N'] = 'A^2'
52      test_wrapper['O'] = 'BA+2C'
53      test_wrapper['P'] = 'E^T'
54      test_wrapper['Q'] = 'C^-1B'
55      test_wrapper['R'] = 'A^{2}3B'
56      test_wrapper['S'] = 'N^-1'
57      test_wrapper['T'] = 'PQP^-1'
58 
59      with pytest.raises(TypeError):
60          test_wrapper['U'] = 'A+1'
61 
62      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'))
```

```python


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'
    assert test_wrapper.get_expression('P') == 'A+C'

    assert test_wrapper.get_expression('Q') == 'N^-1'
    assert test_wrapper.get_expression('R') == 'P-4O'
    assert test_wrapper.get_expression('S') == 'NOP'
```