

lintrans

by D. Dyson

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

Contents

1	Analysis	1
1.1	Computational Approach	1
1.2	Stakeholders	2
1.3	Research on existing solutions	2
1.3.1	MIT ‘Matrix Vector’ Mathlet	2
1.3.2	Linear Transformation Visualizer	3
1.3.3	Desmos app	4
1.3.4	Visualizing Linear Transformations	4
1.4	Essential features	4
1.5	Limitations	5
1.6	Hardware and software requirements	6
1.6.1	Hardware	6
1.6.2	Software	6
1.7	Success criteria	7
2	Design	8
2.1	Problem decomposition	8
2.2	Structure of the solution	8
2.2.1	The main project	8
2.2.2	The guisubpackages	10
2.3	Algorithm design	10
2.4	Usability features	10
2.5	Variables and validation	11
2.6	Iterative test data	12
2.7	Post-development test data	13
3	Development	14
3.1	Matrices backend	14
3.1.1	MatrixWrapperclass	14
3.1.2	Rudimentary parsing and evaluating	16
3.1.3	Simple matrix expression validation	21
3.1.4	Parsing matrix expressions	23
3.2	Initial GUI	26
3.2.1	First basic GUI	26
3.2.2	Numerical definition dialog	28
3.2.3	More definition dialogs	31
3.3	Visualizing matrices	34
3.3.1	Asking strangers on the internet for help	34
3.3.2	Creating the plots package	35
3.3.3	Implementing basis vectors	36
3.3.4	Drawing the transformed grid	38
3.3.5	Implementing animation	41
3.3.6	Preserving determinants	42
3.4	Improving the GUI	44
3.4.1	Fixing rendering	44
3.4.2	Adding vector arrowheads	46
	References	48
A	Project code	49
A.1	__main__.py	49
A.2	__init__.py	50
A.3	typing/__init__.py	50
A.4	matrices/parse.py	51
A.5	matrices/__init__.py	56
A.6	matrices/wrapper.py	57
A.7	gui/main_window.py	61

A.8	gui/validate.py	68
A.9	gui/settings.py	69
A.10	gui/__init__.py	70
A.11	gui/dialogs/define_new_matrix.py	70
A.12	gui/dialogs/misc.py	75
A.13	gui/dialogs/settings.py	77
A.14	gui/dialogs/__init__.py	80
A.15	gui/plots/widgets.py	81
A.16	gui/plots/__init__.py	83
A.17	gui/plots/classes.py	83
B	Testing code	92
B.1	matrices/test_rotation_matrices.py	92
B.2	matrices/test_parse_and_validate_expression.py	93
B.3	matrices/matrix_wrapper/conftest.py	95
B.4	matrices/matrix_wrapper/test_evaluate_expression.py	95
B.5	matrices/matrix_wrapper/test_misc.py	100
B.6	matrices/matrix_wrapper/test_setitem_and_getitem.py	100
B.7	gui/test_dialog_utility_functions.py	103

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[1], 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 be able to gain a visual intuition for linear transformations without worrying about computation until after they've built up that intuition.

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

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

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

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

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

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

1.2 Stakeholders

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

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

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

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

1.3 Research on existing solutions

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

1.3.1 MIT 'Matrix Vector' Mathlet

Arguably the best app that I found was an MIT 'Mathlet' - a simple web app designed to help visualize a maths concept. This one is called 'Matrix Vector'[2] 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.

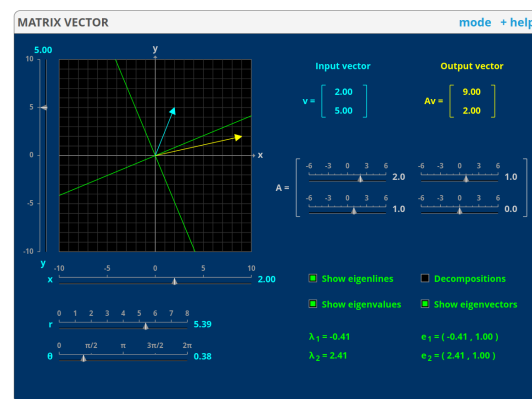


Figure 1: The MIT 'Matrix Vector' Mathlet

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

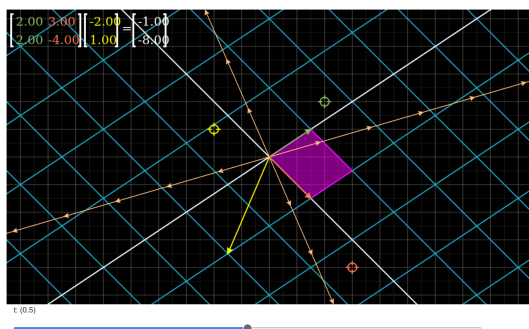


Figure 2: 'Linear Transformation Visualizer' halfway through an animation

Another web app that I found was one simply called 'Linear Transformation Visualizer' by Shad Sharma[3]. 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[4], which was quite hard to use and arguably over-complicated. 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 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

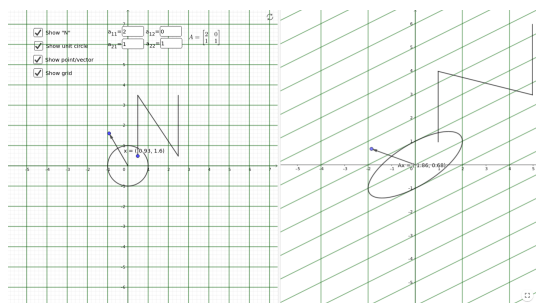


Figure 4: The GeoGebra applet rendering its default matrix

The last solution that I want to talk about is a GeoGebra applet simply titled 'Visualizing Linear Transformations'[5]. This applet has input and output vectors, original and transformed grid lines, a unit circle, and the letter N. It allows the user to define a matrix as 4 numbers and view the aforementioned N (which the user can translate to anywhere on the grid), the unit circle, the input/output vectors, and the grid lines. It also has the input vector snapping to integer coordinates, but that's a standard part of GeoGebra.

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

1.4 Essential features

The primary aim of this application is to visualize linear transformations, so this will obviously be the centre of the app and an essential feature. I will have a widget which can render a background grid

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

Another essential feature is animation. I want the user to be able to smoothly animate between matrices. I see two options for how this could work. If \mathbf{C} is the matrix for the currently displayed transformation, and \mathbf{T} is the matrix for the target transformation, then we could either animate from \mathbf{C} to \mathbf{T} or we could animate from \mathbf{C} to \mathbf{TC} . I would probably call these transitional and applicative animation respectively. Perhaps I'll give the user the option to choose which animation method they want to use. Either way, animation is used in most of the alternative solutions that I found, and it's a great way to build intuition, by allowing students to watch the transformation happen in real time. Compared to simply rendering the transformations, animating them would profoundly benefit learning, and since that's the main aim of the project, I think animation is a necessary part of the app.

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

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

1.5 Limitations

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

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

I'm also not going to do any work with 3D linear transformations. 3D transformations are often harder to visualize and thus it would make sense to target them in an app like this, designed to help with learning and intuition, but 3D transformations are also harder to code. I would have to use a full graphics package rather than a simple widget, and I think it would be too much work for this

project and I wouldn't be able to do it in the time frame. It's definitely a good idea, but I'm currently incapable of creating an app like that.

There are other limitations inherent to matrices. For instance, it's impossible to take an inverse of a singular matrix. There's nothing I can do about that without rewriting most of mathematics. Matrices can also only represent linear transformations. There's definitely a market for an app that could render any arbitrary transformation from $\mathbb{R}^2 \rightarrow \mathbb{R}^2$ - 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 distro². Basically, it just requires an operating system that is compatible with Python 3.10 and Qt5, since I'll be using these in the project. Of course, Qt5 will need to be installed on the user's computer, although it's standard pretty much everywhere these days.

Python 3.10 won't actually be required for the end user, because I will be compiling the app into a stand-alone binary executable for release, and this binary will contain the required Python runtime and dependencies. However, if the user wishes to download and run the source code themselves, then they will need Python 3.10 and the package dependencies: `numpy`, `nptyping`, and `pyqt5`. These can be automatically installed with the command `python -m pip install -r requirements.txt` from the root of the repository. `numpy` is a maths library that allows for fast matrix maths; `nptyping` is used by `mypy` for type-checking and isn't actually a runtime dependency but the imports in the `typing` module

¹Python 3.10 won't compile on any earlier versions of macOS[6]

²Specifying a Linux version is practically impossible. Python 3.10 isn't available in many package repositories, but will compile on any modern distro. Qt5 is available in many package repositories and can be compiled on any x86 or x86_64 generic Linux machine with gcc version 5 or later[7]

fail if it's not installed at runtime; and `pyqt5` is a library that just allows interop between Python and Qt5, which is originally a C++ library.

In the development environment, I use PyCharm for actually writing my code, and I use a virtual environment to isolate my project dependencies. There are also some development dependencies listed in the file `dev_requirements.txt`. They are: `mypy`, `pyqt5-stubs`, `flake8`, `pycodestyle`, `pydocstyle`, and `pytest`. `mypy` is a static type checker³; `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 display information about the currently rendered transformation (determinant, eigenlines, etc.)
6. It must be able to save and load sessions (defined matrices, display settings, etc.)
7. It must allow the user to define and transform arbitrary polygons

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

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

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

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.

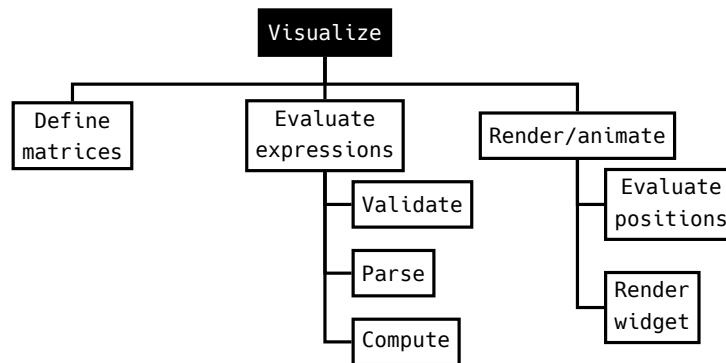
If the majority of teachers would use and/or recommend the app and it meets all of these points, then I will consider the app as a whole to be a success.

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

2 Design

2.1 Problem decomposition

I have decomposed the problem of visualization as follows:



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

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

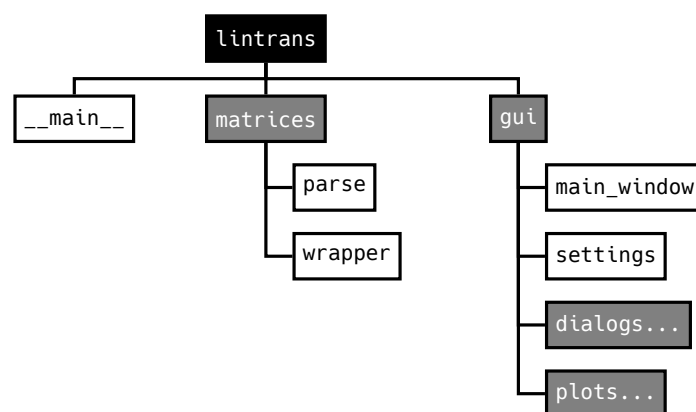
Rendering and animating will likely be the largest part in reality, but I've only decomposed it into simple blocks here. Evaluating positions involves evaluating the matrix expression that the user has input and using the columns of the resultant matrix to find the new positions of the basis vectors,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

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

2.2 Structure of the solution

2.2.1 The main project

I have decomposed my solution like so:



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

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

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

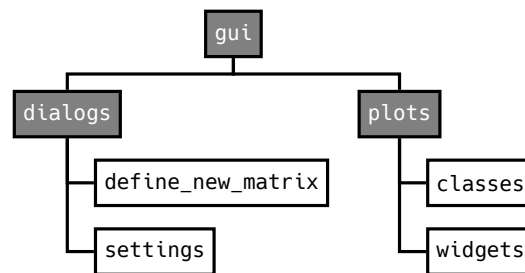
The `settings` module will contain a `DisplaySettings` dataclass⁴ that will represent the settings for visualizing transformations. The `LintransMainWindow` class will have an instance of this class and check against it when rendering things. The user will be able to open a dialog to change these display settings, which will update the main window's instance of this class.

The `settings` module will also have a `GlobalSettings` class, which will represent the global settings for the application, such as the logging level, where to save the logs, whether to ask the user if they want to be prompted with a tutorial whenever they open the app, etc. This class will have defaults for everything, but the constructor will try to read these settings from a config file if possible. This allows for persistent settings between sessions. This config file will be `~/.config/lintrans.conf` on Unix-like systems, including macOS, and `C:\Users\%USER%\AppData\Roaming\lintrans\config.txt` on Windows. This difference is to remain consistent with operating system conventions⁵.

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

⁵And also to avoid confusing Windows users with a `.conf` file

2.2.2 The gui subpackages



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

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

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

2.3 Algorithm design

This section will be completed later.

2.4 Usability features

My main concern in terms of usability is colour. In the 3blue1brown videos on linear algebra, red and green are used for the basis vectors, but these colours are often hard to distinguish in most common forms of colour blindness. The most common form is deuteranopia[8], 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[9] to make sure that my app is accessible to people with different forms of colour blindness⁶.

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

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

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

2.5 Variables and validation

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

The `MatrixWrapper` class will have a dictionary of names and matrices. The names will be single letters⁷ 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* button⁸. 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

⁶I actually had to clone a fork of this project[10] to get it working on Ubuntu 20.04 and adapt it slightly to create a working jar file

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

⁸Actually when the dialog calls `.accept()`. The *Confirm* button is actually connected to a method which first takes the info and updates the instance `MatrixWrapper`, and then calls `.accept()`

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

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

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

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

2.6 Iterative test data

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

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

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

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

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

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

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

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

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

which should parse to give:

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

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

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

2.7 Post-development test data

This section will be completed later.

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 from⁹. 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[12] to allow indexing into a `MatrixWrapper` object like `wrapper['M']`. This simplifies using the class.

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

1  """A module containing a simple MatrixWrapper class to wrap matrices and context."""
2
3  import numpy as np
4
5  from lintrans.typing import MatrixType
6
7
8  class MatrixWrapper:
9      """A simple wrapper class to hold all possible matrices and allow access to them."""
10
11     def __init__(self):
12         """Initialise a MatrixWrapper object with a matrices dict."""
13         self._matrices: dict[str, MatrixType | None] = {
14             'A': None, 'B': None, 'C': None, 'D': None,
15             'E': None, 'F': None, 'G': None, 'H': None,
16             'I': np.eye(2), # I is always defined as the identity matrix
17             'J': None, 'K': None, 'L': None, 'M': None,
18             'N': None, 'O': None, 'P': None, 'Q': None,
19             'R': None, 'S': None, 'T': None, 'U': None,
20             'V': None, 'W': None, 'X': None, 'Y': None,
21             'Z': None
22         }
23
24     def __getitem__(self, name: str) -> MatrixType | None:
25         """Get the matrix with `name` from the dictionary.
26
27         Raises:
28             KeyError:
29                 If there is no matrix with the given name
30         """
31         return self._matrices[name]
32
33     def __setitem__(self, name: str, new_matrix: MatrixType) -> None:
34         """Set the value of matrix `name` with the new_matrix.
35
36         Raises:
37             ValueError:
38                 If `name` isn't a valid matrix name
39         """
40         name = name.upper()
41
42         if name == 'I' or name not in self._matrices:
43             raise NameError('Matrix name must be a capital letter and cannot be "I"')
```

⁹A history of all commits can be found in the GitHub repository[11]

```

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

1  """Test the MatrixWrapper class."""
2
3  import numpy as np
4  import pytest
5  from lintrans.matrices import MatrixWrapper
6
7  valid_matrix_names = 'ABCDEFGHJKLMNPOQRSTUVWXYZ'
8  test_matrix = np.array([[1, 2], [4, 3]])
9
10
11 @pytest.fixture
12 def wrapper() -> MatrixWrapper:
13     """Return a new MatrixWrapper object."""
14     return MatrixWrapper()
15
16
17 def test_get_matrix(wrapper) -> None:
18     """Test MatrixWrapper.__getitem__()."""
19     for name in valid_matrix_names:
20         assert wrapper[name] is None
21
22     assert (wrapper['I'] == np.array([[1, 0], [0, 1]])).all()
23
24
25 def test_get_name_error(wrapper) -> None:
26     """Test that MatrixWrapper.__getitem__() raises a KeyError if called with an invalid name."""
27     with pytest.raises(KeyError):
28         _ = wrapper['bad name']
29         _ = wrapper['123456']
30         _ = wrapper['Th15 Is an 1nV@l1D n@m3']
31         _ = wrapper['abc']
32
33
34 def test_set_matrix(wrapper) -> None:
35     """Test MatrixWrapper.__setitem__()."""
36     for name in valid_matrix_names:
37         wrapper[name] = test_matrix
38         assert (wrapper[name] == test_matrix).all()
39
40
41 def test_set_identity_error(wrapper) -> None:
42     """Test that MatrixWrapper.__setitem__() raises a NameError when trying to assign to I."""
43     with pytest.raises(NameError):
44         wrapper['I'] = test_matrix
45
46
47 def test_set_name_error(wrapper) -> None:
48     """Test that MatrixWrapper.__setitem__() raises a NameError when trying to assign to an invalid name."""
49     with pytest.raises(NameError):
50         wrapper['bad name'] = test_matrix
51         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]?|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 48 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

```

```

1  """Test the MatrixWrapper parse_expression() method."""
2
3  import numpy as np
4  from numpy import linalg as la
5  import pytest
6  from lintrans.matrices import MatrixWrapper
7
8
9  @pytest.fixture
10 def wrapper() -> MatrixWrapper:
11     """Return a new MatrixWrapper object with some preset values."""
12     wrapper = MatrixWrapper()
13
14     root_two_over_two = np.sqrt(2) / 2
15
16     wrapper['A'] = np.array([[1, 2], [3, 4]])
17     wrapper['B'] = np.array([[6, 4], [12, 9]])
18     wrapper['C'] = np.array([[ -1, -3], [4, -12]])
19     wrapper['D'] = np.array([[13.2, 9.4], [-3.4, -1.8]])
20     wrapper['E'] = np.array([
21         [root_two_over_two, -1 * root_two_over_two],
22         [root_two_over_two, root_two_over_two]
23     ])
24     wrapper['F'] = np.array([[ -1, 0], [0, 1]])
25     wrapper['G'] = np.array([[np.pi, np.e], [1729, 743.631]])
26
27     return wrapper
28
29
30 def test_simple_matrix_addition(wrapper: MatrixWrapper) -> None:
31     """Test simple addition and subtraction of two matrices."""
32
33     # NOTE: We assert that all of these values are not None just to stop mypy complaining
34     # These values will never actually be None because they're set in the wrapper() fixture
35     # There's probably a better way do this, because this method is a bit of a bodge, but this works for now
36     assert wrapper['A'] is not None and wrapper['B'] is not None and wrapper['C'] is not None and \
37         wrapper['D'] is not None and wrapper['E'] is not None and wrapper['F'] is not None and \
38         wrapper['G'] is not None
39
40     assert (wrapper.parse_expression('A+B') == wrapper['A'] + wrapper['B']).all()
41     assert (wrapper.parse_expression('E+F') == wrapper['E'] + wrapper['F']).all()
42     assert (wrapper.parse_expression('G+D') == wrapper['G'] + wrapper['D']).all()
43     assert (wrapper.parse_expression('C+C') == wrapper['C'] + wrapper['C']).all()
44     assert (wrapper.parse_expression('D+A') == wrapper['D'] + wrapper['A']).all()
45     assert (wrapper.parse_expression('B+C') == wrapper['B'] + wrapper['C']).all()
46
47
48 def test_simple_two_matrix_multiplication(wrapper: MatrixWrapper) -> None:
49     """Test simple multiplication of two matrices."""
50     assert wrapper['A'] is not None and wrapper['B'] is not None and wrapper['C'] is not None and \
51         wrapper['D'] is not None and wrapper['E'] is not None and wrapper['F'] is not None and \
52         wrapper['G'] is not None
53
54     assert (wrapper.parse_expression('AB') == wrapper['A'] @ wrapper['B']).all()
55     assert (wrapper.parse_expression('BA') == wrapper['B'] @ wrapper['A']).all()
56     assert (wrapper.parse_expression('AC') == wrapper['A'] @ wrapper['C']).all()
57     assert (wrapper.parse_expression('DA') == wrapper['D'] @ wrapper['A']).all()
58     assert (wrapper.parse_expression('ED') == wrapper['E'] @ wrapper['D']).all()
59     assert (wrapper.parse_expression('FD') == wrapper['F'] @ wrapper['D']).all()
60     assert (wrapper.parse_expression('GA') == wrapper['G'] @ wrapper['A']).all()
61     assert (wrapper.parse_expression('CF') == wrapper['C'] @ wrapper['F']).all()
62     assert (wrapper.parse_expression('AG') == wrapper['A'] @ wrapper['G']).all()
63
64
65 def test_identity_multiplication(wrapper: MatrixWrapper) -> None:
66     """Test that multiplying by the identity doesn't change the value of a matrix."""
67     assert wrapper['A'] is not None and wrapper['B'] is not None and wrapper['C'] is not None and \
68         wrapper['D'] is not None and wrapper['E'] is not None and wrapper['F'] is not None and \
69         wrapper['G'] is not None
70
71     assert (wrapper.parse_expression('I') == wrapper['I']).all()
72     assert (wrapper.parse_expression('AI') == wrapper['A']).all()
73     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('IIDI') == 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

9 def compile_valid_expression_pattern() -> Pattern[str]:
10     """Compile the single regular expression that will match a valid matrix expression."""
11     digit_no_zero = '[123456789]'
12     digits = '\\d+'
13     integer_no_zero = '-?' + digit_no_zero + '(' + digits + ')?'
14     real_number = f'({integer_no_zero}(\\.\\{digits}\\)?|-?0?\\.\\{digits}\\)'
15
16     index_content = f'({integer_no_zero}|T)'
17     index = f'\\^\\{index_content}\\}\\^\\{index_content}\\|t)'
18     matrix_identifier = f'([A-Z]|rot\\(\\{real_number}\\))'
19     matrix = '(' + real_number + '?' + matrix_identifier + index + ')?'
20     expression = f'{matrix}+((\\+|-){matrix}+)*'
21
22     return re.compile(expression)
23
24
25 # This is an expensive pattern to compile, so we compile it when this module is initialized
26 valid_expression_pattern = compile_valid_expression_pattern()
27
28
29 def validate_matrix_expression(expression: str) -> bool:
30     """Validate the given matrix expression.
31
32     This function simply checks the expression against a BNF schema. It is not
33     aware of which matrices are actually defined in a wrapper. For an aware
34     version of this function, use the MatrixWrapper().is_valid_expression() method.
35
36     Here is the schema for a valid expression given in a version of BNF:
37
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 );
50
51     :param str expression: The expression to be validated
52     :returns bool: Whether the expression is valid according to the schema
53     """
54     match = valid_expression_pattern.match(expression)
55     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

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

```

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

The `@pytest.mark.parametrize` decorator on line 35 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 33 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):

```

# a453774bcd824676461f9b9b441d7b94969ea55
# 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

1  """The module to provide the main window as a QMainWindow object."""
2
3  import sys
4
5  from PyQt5 import QtCore, QtGui, QtWidgets
6  from PyQt5.QtWidgets import QApplication, QHBoxLayout, QMainWindow, QVBoxLayout
7
8  from lintrans.matrices import MatrixWrapper
9
10
11 class LintransMainWindow(QMainWindow):
12     """The class for the main window in the lintrans GUI."""
13
14     def __init__(self):
15         """Create the main window object, creating every widget in it."""
16         super().__init__()
17
18         self.matrix_wrapper = MatrixWrapper()
19
20         self.setWindowTitle('Linear Transformations')
21         self.setMinimumWidth(750)
22
23         # === Create widgets
24
25         # Left layout: the plot and input box
26
27         # NOTE: This QGraphicsView is only temporary
28         self.plot = QtWidgets.QGraphicsView(self)
29
30         self.text_input_expression = QtWidgets.QLineEdit(self)
31         self.text_input_expression.setPlaceholderText('Input matrix expression...')
32         self.text_input_expression.textChanged.connect(self.update_render_buttons)
33
34         # Right layout: all the buttons
35
36         # Misc buttons
37
38         self.button_create_polygon = QtWidgets.QPushButton(self)
39         self.button_create_polygon.setText('Create polygon')
40         # TODO: Implement create_polygon()
41         # self.button_create_polygon.clicked.connect(self.create_polygon)
42         self.button_create_polygon.setToolTip('Define a new polygon to view the transformation of')
43
44         self.button_change_display_settings = QtWidgets.QPushButton(self)
45         self.button_change_display_settings.setText('Change\ndisplay settings')
46         # TODO: Implement change_display_settings()
47         # self.button_change_display_settings.clicked.connect(self.change_display_settings)
48         self.button_change_display_settings.setToolTip('Change which things are rendered on the plot')
49
50         # Define new matrix buttons
51
```

```

52     self.label_define_new_matrix = QtWidgets.QLabel(self)
53     self.label_define_new_matrix.setText('Define a new 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()

```

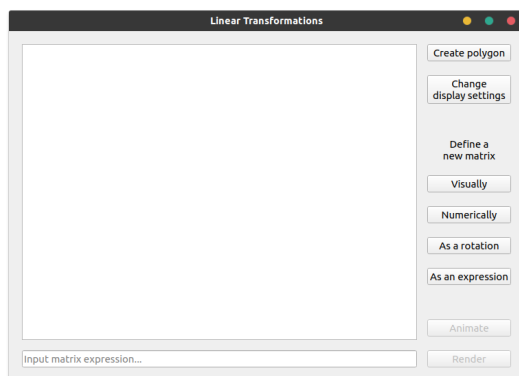


Figure 5: The first version of the GUI

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 35.

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()
```

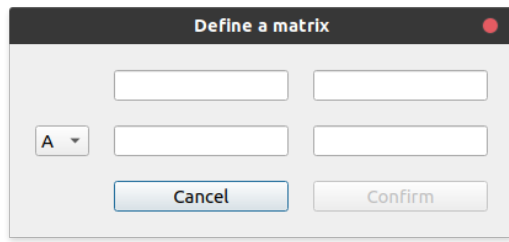


Figure 6: The first version of the numerical definition dialog

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 `DefinedDialog` superclass so that I could easily create other definition dialogs.

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

22 class DefinedDialog(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(DefinedDialog):
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.

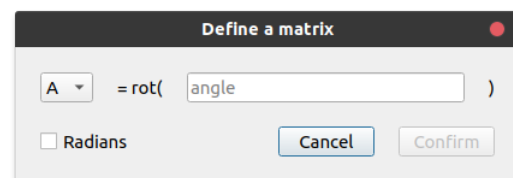


Figure 7: The first version of the rotation definition dialog

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’[13], 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[14].

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[15].

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[16].

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

1  """This module provides the basic classes for plotting transformations."""
2
3  from __future__ import annotations
4
5  from PyQt5.QtCore import Qt
6  from PyQt5.QtGui import QColor, QPainter, QPaintEvent, QPen
7  from PyQt5.QtWidgets import QWidget
8
9
10 class TransformationPlotWidget(QWidget):
11     """An abstract superclass for plot widgets.
12
13     This class provides a background (untransformed) plane, and all the backend
14     details for a Qt application, but does not provide useful functionality. To
15     be useful, this class must be subclassed and behaviour must be implemented
16     by the subclass.
17
18     .. warning:: This class should never be directly instantiated, only subclassed.
19
20     .. note::
21         I would make this class have ``metaclass=abc.ABCMeta``, but I can't because it subclasses ``QWidget``,
22         and a every superclass of a class must have the same metaclass, and ``QWidget`` is not an abstract class.
23     """
24
25     def __init__(self, *args, **kwargs):
26         """Create the widget, passing ``*args`` and ``**kwargs`` to the superclass constructor (``QWidget``)."""
27         super().__init__(*args, **kwargs)
28
29         self.setAutoFillBackground(True)
30
31         # Set the background to white
32         palette = self.palette()
33         palette.setColor(self.backgroundRole(), Qt.white)
34         self.setPalette(palette)
35
36         # Set the grid colour to grey and the axes colour to black
37         self.grid_colour = QColor(128, 128, 128)
38         self.axes_colour = QColor(0, 0, 0)
39
40         self.grid_spacing: int = 50
41         self.line_width: float = 0.4
42
43     @property
44     def w(self) -> int:
45         """Return the width of the widget."""
46         return self.size().width()
47
48     @property
49     def h(self) -> int:
50         """Return the height of the widget."""
51         return self.size().height()
52
53     def paintEvent(self, e: QPaintEvent):
54         """Handle a ``QPaintEvent`` by drawing the widget."""
55         qp = QPainter()
56         qp.begin(self)
57         self.draw_widget(qp)
```

```

58         qp.end()
59
60     def draw_widget(self, qp: QPainter):
61         """Draw the grid and axes in the widget."""
62         qp.setRenderHint(QPainter.Antialiasing)
63         qp.setBrush(Qt.NoBrush)
64
65         # Draw the grid
66         qp.setPen(QPen(self.grid_colour, self.line_width))
67
68         # We draw the background grid, centered in the middle
69         # We deliberately exclude the axes - these are drawn separately
70         for x in range(self.w // 2 + self.grid_spacing, self.w, self.grid_spacing):
71             qp.drawLine(x, 0, x, self.h)
72             qp.drawLine(self.w - x, 0, self.w - x, self.h)
73
74         for y in range(self.h // 2 + self.grid_spacing, self.h, self.grid_spacing):
75             qp.drawLine(0, y, self.w, y)
76             qp.drawLine(0, self.h - y, self.w, self.h - y)
77
78         # Now draw the axes
79         qp.setPen(QPen(self.axes_colour, self.line_width))
80         qp.drawLine(self.w // 2, 0, self.w // 2, self.h)
81         qp.drawLine(0, self.h // 2, self.w, self.h // 2)
82
83
84     class ViewTransformationWidget(TransformationPlotWidget):
85         """This class is used to visualise matrices as transformations."""
86
87     def __init__(self, *args, **kwargs):
88         """Create the widget, passing ``*args`` and ``**kwargs`` to the superclass constructor."""
89         super().__init__(*args, **kwargs)

```

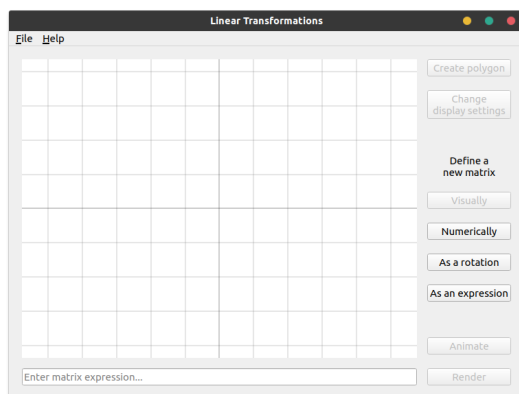


Figure 8: The GUI with background axes

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.

```

# 1fa7e1c61d61cb6aef773b9698541f82fee39ea
# 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.

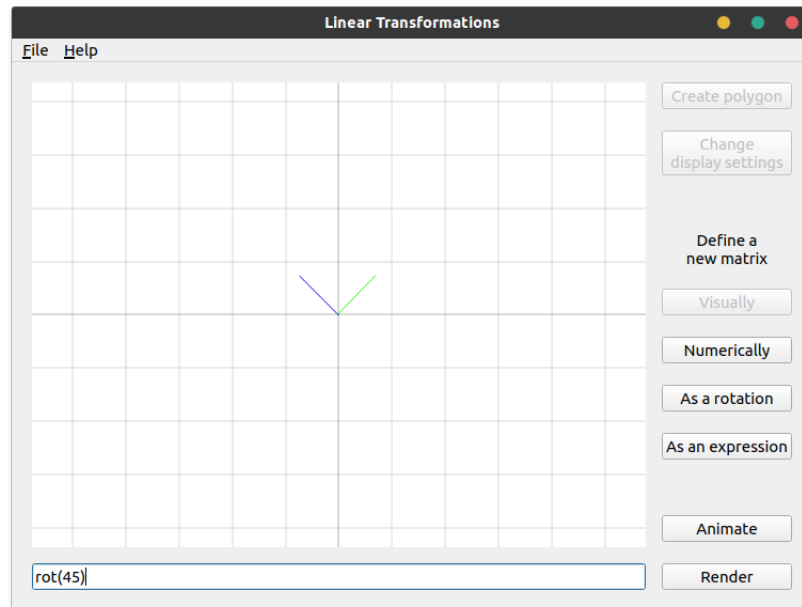


Figure 9: Basis vectors drawn for a 45° rotation

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]) ->
127         None:
128         """Draw a set of grid lines parallel to `vector` intersecting `point`."""
129         max_x, max_y = self.grid_corner()
130         vector_x, vector_y = vector
131         point_x, point_y = point
132
133         if vector_x == 0:
134             painter.drawLine(self.trans_x(0), 0, self.trans_x(0), self.height())

```

```

135         for i in range(int(max_x / point_x)):
136             painter.drawLine(
137                 self.trans_x((i + 1) * point_x),
138                 0,
139                 self.trans_x((i + 1) * point_x),
140                 self.height()
141             )
142             painter.drawLine(
143                 self.trans_x(-1 * (i + 1) * point_x),
144                 0,
145                 self.trans_x(-1 * (i + 1) * point_x),
146                 self.height()
147             )
148
149     elif vector_y == 0:
150         painter.drawLine(0, self.trans_y(0), self.width(), self.trans_y(0))
151
152     for i in range(int(max_y / point_y)):
153         painter.drawLine(
154             0,
155             self.trans_y((i + 1) * point_y),
156             self.width(),
157             self.trans_y((i + 1) * point_y)
158         )
159         painter.drawLine(
160             0,
161             self.trans_y(-1 * (i + 1) * point_y),
162             self.width(),
163             self.trans_y(-1 * (i + 1) * point_y)
164         )

```

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

```

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

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

```

This worked quite well when the matrix involved no rotation, as seen in Figure 10, but this didn't work with rotation. When trying `rot(45)` for example, it looked the same as in Figure 9.

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.

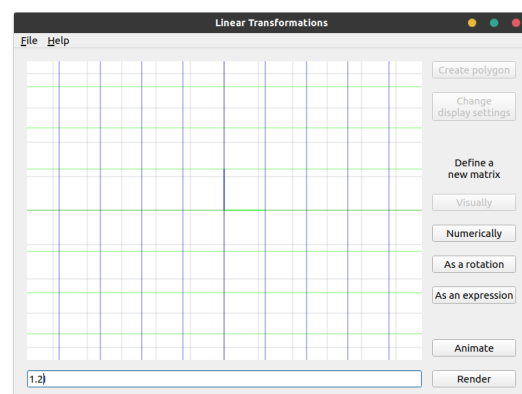


Figure 10: Parallel lines being drawn for matrix $1.2\mathbf{I}$

```

# 7dfe1e24729562501e2fd88a839dca6b653a3375
# 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     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 zero¹⁰ 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.

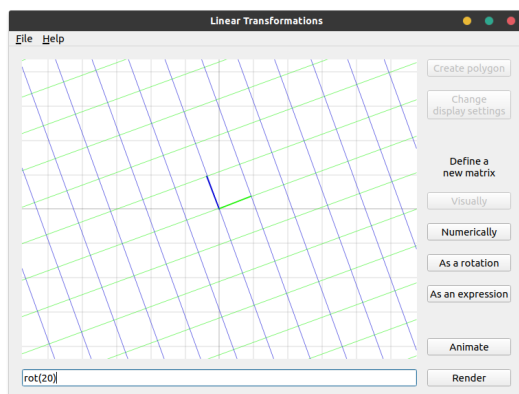


Figure 11: An example of a 20° rotation

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."""

```

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

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

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

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

```

240     self.button_render.setEnabled(False)
241     self.button_animate.setEnabled(False)
242
243     matrix = self.matrix_wrapper.evaluate_expression(self.lineEdit_expression_box.text())
244     matrix_move = matrix - self.matrix_wrapper['I']
245     steps: int = 100
246
247     for i in range(0, steps + 1):
248         self.plot.visualize_matrix_transformation(
249             self.matrix_wrapper['I'] + (i / steps) * matrix_move
250         )
251
252         self.update()
253         self.repaint()
254
255         time.sleep(0.01)
256
257     self.button_render.setEnabled(False)
258     self.button_animate.setEnabled(False)

```

This code creates the `matrix_move` variable and adds scaled versions of it to the identity matrix and renders that each frame. It's simple, but it works well for this simple use case. Unfortunately, it's very hard to show off an animation in a PDF, since all these images are static. The git commit hashes are included in the code snippets if you want to clone the repo[11], 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.

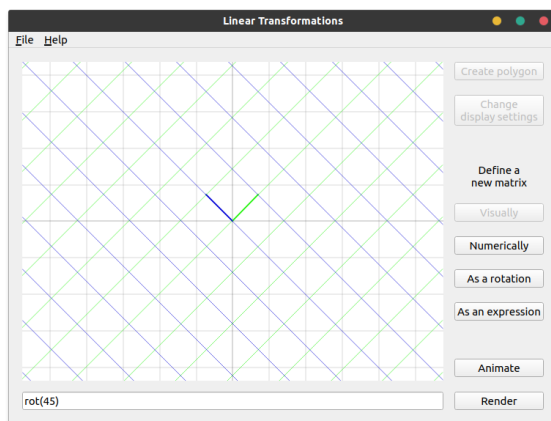


Figure 12: What we would expect halfway through a 90° rotation

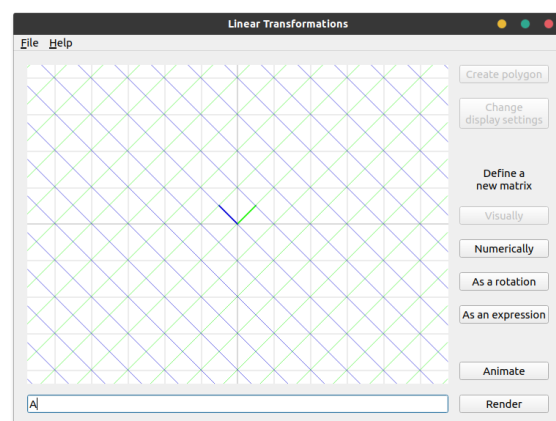


Figure 13: What we actually get halfway through a 90° rotation

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[17] 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.linedit_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 do 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.

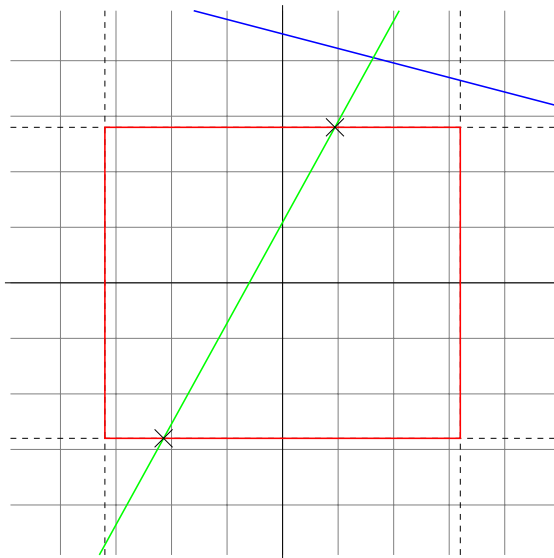


Figure 14: Two example lines and the viewport box

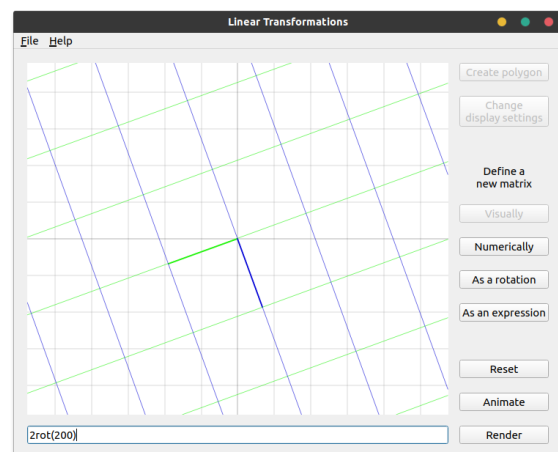


Figure 15: A demonstration of the new oblique lines system.

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

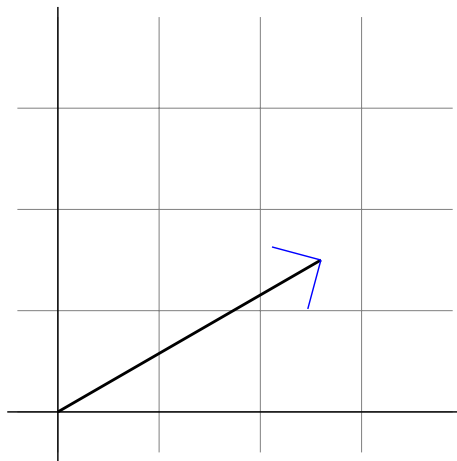


Figure 16: An example of a vector with the arrowheads highlighted in green

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 csharpshelper.com¹¹, 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://csharpshelper.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))
```

¹¹This website is currently being rewritten and this arrowheads tutorial is listed as 'not rebuilt' on <http://csharpshelper.com>

```

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()`.

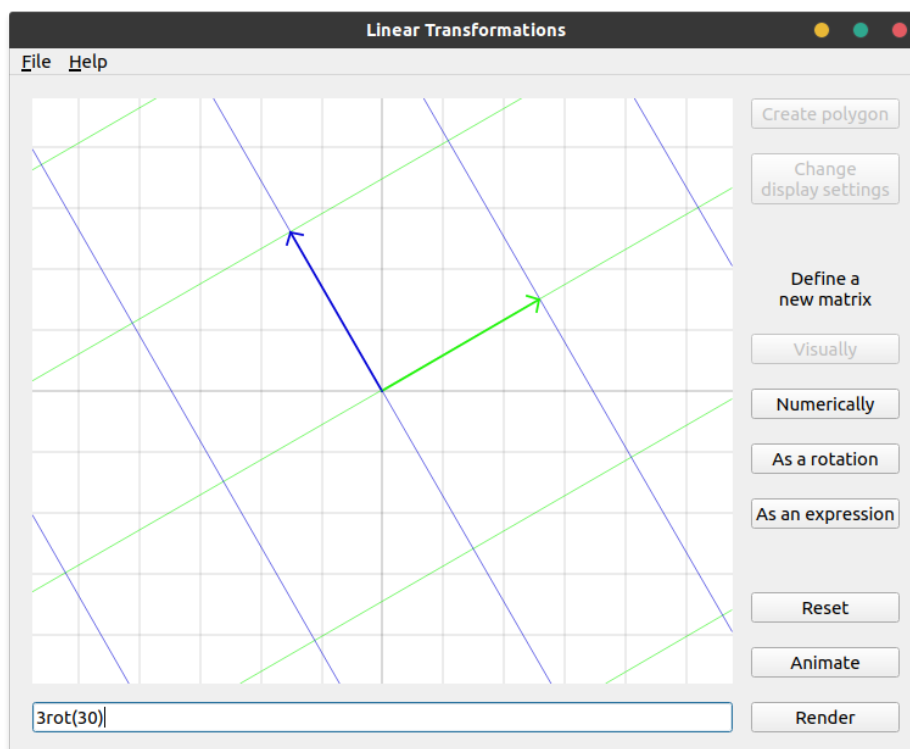


Figure 17: An example of the i and j vectors with arrowheads

References

- [1] Grant Sanderson (3blue1brown). *Essence of Linear Algebra*. 6th Aug. 2016. URL: https://www.youtube.com/playlist?list=PLZHQ0b0WTQDPD3MizzM2xVFitgF8hE_ab.
- [2] H. Hohn et al. *Matrix Vector*. MIT. 2001. URL: <https://mathlets.org/mathlets/matrix-vector/>.
- [3] Shad Sharma. *Linear Transformation Visualizer*. 4th May 2017. URL: <https://shad.io/MatVis/>.
- [4] *2D linear transformation*. URL: <https://www.desmos.com/calculator/upooihuy4s>.
- [5] je1324. *Visualizing Linear Transformations*. 15th Mar. 2018. URL: <https://www.geogebra.org/m/YCZa8TAH>.
- [6] *Python 3.10 Downloads*. Python Software Foundation. URL: <https://www.python.org/downloads/release/python-3100/>.
- [7] *Qt5 for Linux/X11*. URL: <https://doc.qt.io/qt-5/linux.html>.
- [8] *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>.
- [9] Nathaniel Vaughn Kelso and Bernie Jenny. *Color Oracle*. Version 1.3. URL: <https://colororacle.org/>.
- [10] Alanocallaghan. *color-oracle-java*. Version 1.3. URL: <https://github.com/Alanocallaghan/color-oracle-java>.
- [11] D. Dyson (DoctorDalek1963). *lintrans*. GitHub. URL: <https://github.com/DoctorDalek1963/lintrans>.
- [12] *Python 3 Data model - special methods*. Python Software Foundation. URL: <https://docs.python.org/3/reference/datamodel.html#special-method-names>.
- [13] *PyQt5 Graphics View Framework*. URL: <https://doc.qt.io/qtforpython-5/overviews/graphicsview.html>.
- [14] *Plotting with Matplotlib*. URL: <https://www.pythonguis.com/tutorials/plotting-matplotlib/>.
- [15] D. Dyson. *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>.
- [16] Ross Wilson. *Python-Etudes/PyQtCustomWidget*. URL: <https://gitlab.com/rzzzwilson/python-etudes/-/tree/master/PyQtCustomWidget>.
- [17] *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.
- [18] *Draw lines with arrowheads in C#*. 5th Dec. 2014. URL: <http://csharp-helper.com/blog/2014/12/draw-lines-with-arrowheads-in-c/>.

A Project code

A.1 __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  import sys
12  from argparse import ArgumentParser
13  from textwrap import dedent
14
15  from lintrans import __version__
16  from lintrans.gui import main_window
17
18
19  def main(args: list[str]) -> None:
20      """Interpret program-specific command line arguments and run the main window in most cases.
21
22      If the user supplies --help or --version, then we simply respond to that and then return.
23      If they don't supply either of these, then we run :func:`lintrans.gui.main_window.main`.
24
25      :param list[str] args: The full argument list (including program name)
26      """
27      parser = ArgumentParser(add_help=False)
28
29      parser.add_argument(
30          '-h',
31          '--help',
32          default=False,
33          action='store_true'
34      )
35
36      parser.add_argument(
37          '-V',
38          '--version',
39          default=False,
40          action='store_true'
41      )
42
43      parsed_args, unparsed_args = parser.parse_known_args()
44
45      if parsed_args.help:
46          print(dedent(''
47              Usage: lintrans [option]
48
49              Options:
50                  -h, --help      Display this help text and exit
51                  -V, --version   Display the version information and exit
52
53              Any other options will get passed to the QApplication constructor.
54              If you don't know what that means, then don't provide any arguments and just the run the program.''[1:]))
55          return
56
57      if parsed_args.version:
58          print(dedent(f''
59              lintrans (version {__version__})
60              The linear transformation visualizer
61
62              Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
63
64              This program is licensed under GNU GPLv3, available here:
65              <https://www.gnu.org/licenses/gpl-3.0.html>''[1:]))
66          return
67

```

```

68     for arg in unparsed_args:
69         print(f'Passing "{arg}" to QApplication. See --help for recognised args')
70
71     main_window.main(args[:1] + unparsed_args)
72
73
74 if __name__ == '__main__':
75     main(sys.argv)

```

A.2 __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 gui, matrices, typing_
10
11  __version__ = '0.2.2-alpha'
12
13  __all__ = ['gui', 'matrices', 'typing_', '__version__']

```

A.3 typing_/__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 type aliases for linear algebra and transformations.
8
9  .. note::
10     This package is called ``typing_`` and not ``typing`` to avoid name collisions with the
11     builtin :mod:`typing`. I don't quite know how this collision occurs, but renaming
12     this module fixed the problem.
13 """
14
15 from __future__ import annotations
16
17 from typing import Any, TypeGuard
18
19 from numpy import ndarray
20 from nptyping import NDArray, Float
21
22 __all__ = ['is_matrix_type', 'MatrixType', 'MatrixParseList']
23
24 MatrixType = NDArray[(2, 2), Float]
25 """This type represents a 2x2 matrix as a NumPy array."""
26
27 MatrixParseList = list[list[tuple[str, str, str]]]
28 """This is a list containing lists of tuples. Each tuple represents a matrix and is ``(multiplier,
29 matrix_identifier, index)`` where all of them are strings. These matrix-representing tuples are
30 contained in lists which represent multiplication groups. Every matrix in the group should be
31 multiplied together, in order. These multiplication group lists are contained by a top level list,
32 which is this type. Once these multiplication group lists have been evaluated, they should be summed.
33
34 In the tuples, the multiplier is a string representing a real number, the matrix identifier
35 is a capital letter or ``rot(x)`` where x is a real number angle, and the index is a string
36 representing an integer, or it's the letter ``T`` for transpose.
37 """
38
39
40 def is_matrix_type(matrix: Any) -> TypeGuard[NDArray[(2, 2), Float]]:
41     """Check if the given value is a valid matrix type.

```

```

42
43     .. note::
44         This function is a TypeGuard, meaning if it returns True, then the
45         passed value must be a :attr:`lintrans.typing.MatrixType`.
46     """
47     return isinstance(matrix, ndarray) and matrix.shape == (2, 2)

```

A.4 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 Pattern
14
15  from lintrans.typing import MatrixParseList
16
17  NAIVE_CHARACTER_CLASS = r'[-+\sA-Z0-9.rot()^{}]'
18
19
20  class MatrixParseError(Exception):
21      """A simple exception to be raised when an error is found when parsing."""
22
23
24  def compile_naive_expression_pattern() -> Pattern[str]:
25      """Compile the single RegEx pattern that will match a valid matrix expression."""
26      digit_no_zero = '[123456789]'
27      digits = '\\d+'
28      integer_no_zero = digit_no_zero + '(' + digits + ')?'
29      real_number = f'({integer_no_zero}(\\.({digits})?|0\\.({digits}))'
30
31      index_content = f'(-?{integer_no_zero}|T)'
32      index = f'\\^{{{index_content}}}|\\^{{{index_content}}}'
33      matrix_identifier = f'([A-Z]|rot\\((-?{real_number}\\)|\\({NAIVE_CHARACTER_CLASS}\\}\\))'
34      matrix = '(' + real_number + '?' + matrix_identifier + index + '?'
35      expression = f'^~?{matrix}+((\\+~?|-){matrix}+)*$'
36
37      return re.compile(expression)
38
39
40  # This is an expensive pattern to compile, so we compile it when this module is initialized
41  naive_expression_pattern = compile_naive_expression_pattern()
42
43
44  def find_sub_expressions(expression: str) -> list[str]:
45      """Find all the sub-expressions in the given expression.
46
47      This function only goes one level deep, so may return strings like ``A(BC)D``.
48
49      :raises MatrixParseError: If there are unbalanced parentheses
50      """
51      sub_expressions: list[str] = []
52      string = ''
53      paren_depth = 0
54      pointer = 0
55
56      while True:
57          char = expression[pointer]
58
59          if char == '(' and expression[pointer - 3:pointer] != 'rot':
60              paren_depth += 1
61

```

```

62         # This is a bit of a manual bodge, but it eliminates extraneous parens
63         if paren_depth == 1:
64             pointer += 1
65             continue
66
67         elif char == ')' and re.match(f'[{NAIVE_CHARACTER_CLASS}]*?rot\\([-\d.]+\$', expression[:pointer]) is None:
68             paren_depth -= 1
69
70         if paren_depth > 0:
71             string += char
72
73         if paren_depth == 0 and string:
74             sub_expressions.append(string)
75             string = ''
76
77         pointer += 1
78
79         if pointer >= len(expression):
80             break
81
82     if paren_depth != 0:
83         raise MatrixParseError('Unbalanced parentheses in expression')
84
85     return sub_expressions
86
87
88 def validate_matrix_expression(expression: str) -> bool:
89     """Validate the given matrix expression.
90
91     This function simply checks the expression against the BNF schema documented in
92     :ref:`expression-syntax-docs`. It is not aware of which matrices are actually defined
93     in a wrapper. For an aware version of this function, use the
94     :meth:`lintrans.matrices.wrapper.MatrixWrapper.is_valid_expression` method.
95
96     :param str expression: The expression to be validated
97     :returns bool: Whether the expression is valid according to the schema
98     """
99     # Remove all whitespace
100    expression = re.sub(r'\s', '', expression)
101
102    match = naive_expression_pattern.match(expression)
103
104    if match is None:
105        return False
106
107    # Check that the whole expression was matched against
108    if expression != match.group(0):
109        return False
110
111    try:
112        sub_expressions = find_sub_expressions(expression)
113    except MatrixParseError:
114        return False
115
116    if not sub_expressions:
117        return True
118
119    return all(validate_matrix_expression(m) for m in sub_expressions)
120
121
122 @dataclass
123 class MatrixToken:
124     """A simple dataclass to hold information about a matrix token being parsed."""
125
126     multiplier: str = ''
127     identifier: str = ''
128     exponent: str = ''
129
130     @property
131     def tuple(self) -> tuple[str, str, str]:
132         """Create a tuple of the token for parsing."""
133         return self.multiplier, self.identifier, self.exponent
134

```

```

135
136 class ExpressionParser:
137     """A class to hold state during parsing.
138
139     Most of the methods in this class are class-internal and should not be used from outside.
140
141     This class should be used like this:
142
143     >>> ExpressionParser('3A~1B').parse()
144     [[('3', 'A', '-1'), ('', 'B', '')]]
145     >>> ExpressionParser('4(M^TA^2)^~2').parse()
146     [[('4', 'M^{T}A^{2}', '-2')]]
147     """
148
149     def __init__(self, expression: str):
150         """Create an instance of the parser with the given expression and initialise variables to use during
151         ↪ parsing."""
152         # Remove all whitespace
153         expression = re.sub(r'\s', '', expression)
154
155         # Check if it's valid
156         if not validate_matrix_expression(expression):
157             raise MatrixParseError('Invalid expression')
158
159         # Wrap all exponents and transposition powers with {}
160         expression = re.sub(r'(<=^)(-?\d+|T)(?=[^}]|$)', r'{\g<0>}', expression)
161
162         # Remove any standalone minuses
163         expression = re.sub(r'-(?=[A-Z])', '-1', expression)
164
165         # Replace subtractions with additions
166         expression = re.sub(r'-(?=\d+\.?\d*([A-Z]|rot))', '+-', expression)
167
168         # Get rid of a potential leading + introduced by the last step
169         expression = re.sub(r'^+', '', expression)
170
171         self.expression = expression
172         self.pointer: int = 0
173
174         self.current_token = MatrixToken()
175         self.current_group: list[tuple[str, str, str]] = []
176
177         self.final_list: MatrixParseList = []
178
179     def __repr__(self) -> str:
180         """Return a simple repr containing the expression."""
181         return f'{self.__class__.__module__}.{self.__class__.__name__}("{self.expression}")'
182
183     @property
184     def char(self) -> str:
185         """Return the char pointed to by the pointer."""
186         return self.expression[self.pointer]
187
188     def parse(self) -> MatrixParseList:
189         """Fully parse the instance's matrix expression and return the :attr:`lintrans.typing.MatrixParseList`.
190
191         This method uses all the private methods of this class to parse the
192         expression in parts. All private methods mutate the instance variables.
193
194         :returns: The parsed expression
195         :rtype: :attr:`lintrans.typing.MatrixParseList`
196         """
197         self._parse_multiplication_group()
198
199         while self.pointer < len(self.expression):
200             if self.expression[self.pointer] != '+':
201                 raise MatrixParseError('Expected "+" between multiplication groups')
202
203             self.pointer += 1
204             self._parse_multiplication_group()
205
206         return self.final_list

```



```

207 def _parse_multiplication_group(self) -> None:
208     """Parse a group of matrices to be multiplied together.
209
210     This method just parses matrices until we get to a ``+``.
211     """
212     # This loop continues to parse matrices until we fail to do so
213     while self._parse_matrix():
214         # Once we get to the end of the multiplication group, we add it the final list and reset the group list
215         if self.pointer >= len(self.expression) or self.char == '+':
216             self.final_list.append(self.current_group)
217             self.current_group = []
218             self.pointer += 1
219
220 def _parse_matrix(self) -> bool:
221     """Parse a full matrix using :meth:`_parse_matrix_part`.
222
223     This method will parse an optional multiplier, an identifier, and an optional exponent. If we
224     do this successfully, we return True. If we fail to parse a matrix (maybe we've reached the
225     end of the current multiplication group and the next char is ``+``), then we return False.
226
227     :returns bool: Success or failure
228     """
229     self.current_token = MatrixToken()
230
231     while self._parse_matrix_part():
232         pass # The actual execution is taken care of in the loop condition
233
234     if self.current_token.identifier == '':
235         return False
236
237     self.current_group.append(self.current_token.tuple)
238     return True
239
240 def _parse_matrix_part(self) -> bool:
241     """Parse part of a matrix (multiplier, identifier, or exponent).
242
243     Which part of the matrix we parse is dependent on the current value of the pointer and the expression.
244     This method will parse whichever part of matrix token that it can. If it can't parse a part of a matrix,
245     or it's reached the next matrix, then we just return False. If we succeeded to parse a matrix part, then
246     we return True.
247
248     :returns bool: Success or failure
249     :raises MatrixParseError: If we fail to parse this part of the matrix
250     """
251     if self.pointer >= len(self.expression):
252         return False
253
254     if self.char.isdigit() or self.char == '-':
255         if self.current_token.multiplier != '' \
256             or (self.current_token.multiplier == '' and self.current_token.identifier != ''):
257             return False
258
259         self._parse_multiplier()
260
261     elif self.char.isalpha() and self.char.isupper():
262         if self.current_token.identifier != '':
263             return False
264
265         self.current_token.identifier = self.char
266         self.pointer += 1
267
268     elif self.char == 'r':
269         if self.current_token.identifier != '':
270             return False
271
272         self._parse_rot_identifier()
273
274     elif self.char == '(':
275         if self.current_token.identifier != '':
276             return False
277
278         self._parse_sub_expression()
279

```

```

280         elif self.char == '^':
281             if self.current_token.exponent != '':
282                 return False
283
284             self._parse_exponent()
285
286         elif self.char == '+':
287             return False
288
289         else:
290             raise MatrixParseError(f'Unrecognised character "{self.char}" in matrix expression')
291
292         return True
293
294     def _parse_multiplier(self) -> None:
295         """Parse a multiplier from the expression and pointer.
296
297         This method just parses a numerical multiplier, which can include
298         zero or one ``.`` character and optionally a ``-`` at the start.
299
300         :raises MatrixParseError: If we fail to parse this part of the matrix
301         """
302         multiplier = ''
303
304         while self.char.isdigit() or self.char in ('.', '-'):
305             multiplier += self.char
306             self.pointer += 1
307
308         try:
309             float(multiplier)
310         except ValueError as e:
311             raise MatrixParseError(f'Invalid multiplier "{multiplier}"') from e
312
313         self.current_token.multiplier = multiplier
314
315     def _parse_rot_identifier(self) -> None:
316         """Parse a ``rot()``-style identifier from the expression and pointer.
317
318         This method will just parse something like ``rot(12.5)``. The angle number must be a real number.
319
320         :raises MatrixParseError: If we fail to parse this part of the matrix
321         """
322         if match := re.match(r'rot\(((\d.-]+\))\)', self.expression[self.pointer:]):
323             # Ensure that the number in brackets is a valid float
324             try:
325                 float(match.group(1))
326             except ValueError as e:
327                 raise MatrixParseError(f'Invalid angle number "{match.group(1)}" in rot-identifier') from e
328
329             self.current_token.identifier = match.group(0)
330             self.pointer += len(match.group(0))
331         else:
332             raise MatrixParseError(f'Invalid rot-identifier "{self.expression[self.pointer:self.pointer + 15]}'
333             }...")
334
335     def _parse_sub_expression(self) -> None:
336         """Parse a parenthesized sub-expression as the identifier.
337
338         This method will also validate the expression in the parentheses.
339
340         :raises MatrixParseError: If we fail to parse this part of the matrix
341         """
342         if self.char != '(':
343             raise MatrixParseError('Sub-expression must start with "("')
344
345         self.pointer += 1
346         paren_depth = 1
347         identifier = ''
348
349         while paren_depth > 0:
350             if self.char == '(':
351                 paren_depth += 1
352             elif self.char == ')':

```

```

352         paren_depth -= 1
353
354     if paren_depth == 0:
355         self.pointer += 1
356         break
357
358     identifier += self.char
359     self.pointer += 1
360
361     if not validate_matrix_expression(identifier):
362         raise MatrixParseError(f'Invalid sub-expression identifier "{identifier}"')
363
364     self.current_token.identifier = identifier
365
366 def _parse_exponent(self) -> None:
367     """Parse a matrix exponent from the expression and pointer.
368
369     The exponent must be an integer or ``T`` for transpose.
370
371     :raises MatrixParseError: If we fail to parse this part of the token
372     """
373     if match := re.match(r'\^{\d+|T}\}', self.expression[self.pointer:]):
374         exponent = match.group(1)
375
376         try:
377             if exponent != 'T':
378                 int(exponent)
379         except ValueError as e:
380             raise MatrixParseError(f'Invalid exponent "{match.group(1)}"') from e
381
382         self.current_token.exponent = exponent
383         self.pointer += len(match.group(0))
384     else:
385         raise MatrixParseError(f'Invalid exponent "{self.expression[self.pointer:self.pointer + 10]}..."')
386
387
388 def parse_matrix_expression(expression: str) -> MatrixParseList:
389     """Parse the matrix expression and return a :data:`lintrans.typing_.MatrixParseList`.
390
391     :Example:
392
393     >>> parse_matrix_expression('A')
394     [[(' ', 'A', ' ')]]
395     >>> parse_matrix_expression('-3M^2')
396     [[(' ', 'M', '2')]]
397     >>> parse_matrix_expression('1.2rot(12)^{3}2B^T')
398     [[('1.2', 'rot(12)', '3'), ('2', 'B', 'T')]]
399     >>> parse_matrix_expression('A^2 + 3B')
400     [[(' ', 'A', '2')], [('3', 'B', ' ')]]
401     >>> parse_matrix_expression('-3A^{-1}3B^T - 45M^2')
402     [[(' ', 'A', '-1'), ('3', 'B', 'T')], [(' ', 'M', '2')]]
403     >>> parse_matrix_expression('5.3A^{4} 2.6B^{-2} + 4.6D^T 8.9E^{-1}')
404     [[('5.3', 'A', '4'), ('2.6', 'B', '-2')], [('4.6', 'D', 'T'), ('8.9', 'E', '-1')]]
405     >>> parse_matrix_expression('2(A+B^TC)^2D')
406     [[('2', 'A+B^TC', '2'), (' ', 'D', ' ')]]
407
408     :param str expression: The expression to be parsed
409     :returns: A list of parsed components
410     :rtype: :data:`lintrans.typing_.MatrixParseList`
411     """
412     return ExpressionParser(expression).parse()

```

A.5 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
10 from .wrapper import create_rotation_matrix, MatrixWrapper
11
12 __all__ = ['create_rotation_matrix', 'MatrixWrapper', 'parse']

```

A.6 matrices/wrapper.py

```

1  # lintrans - The linear transformation visualizer
2  # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4  # This program is licensed under GNU GPLv3, available here:
5  # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7  """This module contains the main :class:`MatrixWrapper` class and a function to create a matrix from an angle."""
8
9  from __future__ import annotations
10
11 import re
12 from copy import copy
13 from functools import reduce
14 from operator import add, matmul
15 from typing import Any, Optional, Union
16
17 import numpy as np
18
19 from .parse import parse_matrix_expression, validate_matrix_expression
20 from lintrans.typing import is_matrix_type, MatrixType
21
22
23 class MatrixWrapper:
24     """A wrapper class to hold all possible matrices and allow access to them.
25
26     .. note::
27         When defining a custom matrix, its name must be a capital letter and cannot be ``I``.
28
29     The contained matrices can be accessed and assigned to using square bracket notation.
30
31     :Example:
32
33     >>> wrapper = MatrixWrapper()
34     >>> wrapper['I']
35     array([[1., 0.],
36           [0., 1.]])
37     >>> wrapper['M'] # Returns None
38     >>> wrapper['M'] = np.array([[1, 2], [3, 4]])
39     >>> wrapper['M']
40     array([[1., 2.],
41           [3., 4.]])
42     """
43
44     def __init__(self):
45         """Initialize a :class:`MatrixWrapper` object with a dictionary of matrices which can be accessed."""
46         self._matrices: dict[str, Optional[Union[MatrixType, str]]] = {
47             'A': None, 'B': None, 'C': None, 'D': None,
48             'E': None, 'F': None, 'G': None, 'H': None,
49             'I': np.eye(2), # I is always defined as the identity matrix
50             'J': None, 'K': None, 'L': None, 'M': None,
51             'N': None, 'O': None, 'P': None, 'Q': None,
52             'R': None, 'S': None, 'T': None, 'U': None,
53             'V': None, 'W': None, 'X': None, 'Y': None,
54             'Z': None
55         }
56
57     def __repr__(self) -> str:
58         """Return a nice string repr of the :class:`MatrixWrapper` for debugging."""
59         defined_matrices = ''.join([k for k, v in self._matrices.items() if v is not None])
60         return f'<{self.__class__.__module__}.{self.__class__.__name__} object with ' \
61             f'{len(defined_matrices)} defined matrices: '{defined_matrices}'>

```

```

62
63 def __eq__(self, other: Any) -> bool:
64     """Check for equality in wrappers by comparing dictionaries.
65
66     :param Any other: The object to compare this wrapper to
67     """
68     if not isinstance(other, self.__class__):
69         return NotImplemented
70
71     # We loop over every matrix and check if every value is equal in each
72     for name in self._matrices:
73         s_matrix = self[name]
74         o_matrix = other[name]
75
76         if s_matrix is None and o_matrix is None:
77             continue
78
79         elif (s_matrix is None and o_matrix is not None) or \
80              (s_matrix is not None and o_matrix is None):
81             return False
82
83         # This is mainly to satisfy mypy, because we know these must be matrices
84         elif not is_matrix_type(s_matrix) or not is_matrix_type(o_matrix):
85             return False
86
87         # Now we know they're both NumPy arrays
88         elif np.array_equal(s_matrix, o_matrix):
89             continue
90
91         else:
92             return False
93
94     return True
95
96 def __hash__(self) -> int:
97     """Return the hash of the matrices dictionary."""
98     return hash(self._matrices)
99
100 def __getitem__(self, name: str) -> Optional[MatrixType]:
101     """Get the matrix with the given name.
102
103     If it is a simple name, it will just be fetched from the dictionary. If the name is ``rot(x)`, with
104     a given angle in degrees, then we return a new matrix representing a rotation by that angle.
105
106     .. note::
107         If the named matrix is defined as an expression, then this method will return its evaluation.
108         If you want the expression itself, use :meth:`get_expression`.
109
110     :param str name: The name of the matrix to get
111     :returns Optional[MatrixType]: The value of the matrix (could be None)
112
113     :raises NameError: If there is no matrix with the given name
114     """
115     # Return a new rotation matrix
116     if (match := re.match(r'^rot\((-?\d*\.\d*)\)$', name)) is not None:
117         return create_rotation_matrix(float(match.group(1)))
118
119     if name not in self._matrices:
120         if validate_matrix_expression(name):
121             return self.evaluate_expression(name)
122
123         raise NameError(f'Unrecognised matrix name "{name}"')
124
125     # We copy the matrix before we return it so the user can't accidentally mutate the matrix
126     matrix = copy(self._matrices[name])
127
128     if isinstance(matrix, str):
129         return self.evaluate_expression(matrix)
130
131     return matrix
132
133 def __setitem__(self, name: str, new_matrix: Optional[Union[MatrixType, str]]) -> None:
134     """Set the value of matrix ``name`` with the new_matrix.

```

```

135
136     The new matrix may be a simple 2x2 NumPy array, or it could be a string, representing an
137     expression in terms of other, previously defined matrices.
138
139     :param str name: The name of the matrix to set the value of
140     :param Optional[Union[MatrixType, str]] new_matrix: The value of the new matrix (could be None)
141
142     :raises NameError: If the name isn't a legal matrix name
143     :raises TypeError: If the matrix isn't a valid 2x2 NumPy array or expression in terms of other defined
144     ↪ matrices
145     :raises ValueError: If you attempt to define a matrix in terms of itself
146     """
147     if not (name in self._matrices and name != 'I'):
148         raise NameError('Matrix name is illegal')
149
150     if new_matrix is None:
151         self._matrices[name] = None
152         return
153
154     if isinstance(new_matrix, str):
155         if self.is_valid_expression(new_matrix):
156             if name not in self._matrices:
157                 self._matrices[name] = new_matrix
158                 return
159             else:
160                 raise ValueError('Cannot define a matrix recursively')
161
162     if not is_matrix_type(new_matrix):
163         raise TypeError('Matrix must be a 2x2 NumPy array')
164
165     # All matrices must have float entries
166     a = float(new_matrix[0][0])
167     b = float(new_matrix[0][1])
168     c = float(new_matrix[1][0])
169     d = float(new_matrix[1][1])
170
171     self._matrices[name] = np.array([[a, b], [c, d]])
172
173 def get_expression(self, name: str) -> Optional[str]:
174     """If the named matrix is defined as an expression, return that expression, else return None.
175
176     :param str name: The name of the matrix
177     :returns Optional[str]: The expression that the matrix is defined as, or None
178
179     :raises NameError: If the name is invalid
180     """
181     if name not in self._matrices:
182         raise NameError('Matrix must have a legal name')
183
184     matrix = self._matrices[name]
185     if isinstance(matrix, str):
186         return matrix
187
188     return None
189
190 def is_valid_expression(self, expression: str) -> bool:
191     """Check if the given expression is valid, using the context of the wrapper.
192
193     This method calls :func:`lintrans.matrices.parse.validate_matrix_expression`, but also
194     ensures that all the matrices in the expression are defined in the wrapper.
195
196     :param str expression: The expression to validate
197     :returns bool: Whether the expression is valid in this wrapper
198
199     :raises LinAlgError: If a matrix is defined in terms of the inverse of a singular matrix
200     """
201     # Get rid of the transposes to check all capital letters
202     new_expression = expression.replace('^T', '').replace('^{T}', '')
203
204     # Make sure all the referenced matrices are defined
205     for matrix in [x for x in new_expression if re.match('[A-Z]', x)]:
206         if self[matrix] is None:
207             return False

```

```

207
208         if (expr := self.get_expression(matrix)) is not None:
209             if not self.is_valid_expression(expr):
210                 return False
211
212     return validate_matrix_expression(expression)
213
214 def evaluate_expression(self, expression: str) -> MatrixType:
215     """Evaluate a given expression and return the matrix evaluation.
216
217     :param str expression: The expression to be parsed
218     :returns MatrixType: The matrix result of the expression
219
220     :raises ValueError: If the expression is invalid
221     """
222     if not self.is_valid_expression(expression):
223         raise ValueError('The expression is invalid')
224
225     parsed_result = parse_matrix_expression(expression)
226     final_groups: list[list[MatrixType]] = []
227
228     for group in parsed_result:
229         f_group: list[MatrixType] = []
230
231         for multiplier, identifier, index in group:
232             if index == 'T':
233                 m = self[identifier]
234
235                 # This assertion is just so mypy doesn't complain
236                 # We know this won't be None, because we know that this matrix is defined in this wrapper
237                 assert m is not None
238                 matrix_value = m.T
239
240             else:
241                 matrix_value = np.linalg.matrix_power(self[identifier], 1 if index == '1' else int(index))
242
243                 matrix_value *= 1 if multiplier == '1' else float(multiplier)
244                 f_group.append(matrix_value)
245
246         final_groups.append(f_group)
247
248     return reduce(add, [reduce(matmul, group) for group in final_groups])
249
250
251 def create_rotation_matrix(angle: float, *, degrees: bool = True) -> MatrixType:
252     """Create a matrix representing a rotation (anticlockwise) by the given angle.
253
254     :Example:
255
256     >>> create_rotation_matrix(30)
257     array([[ 0.8660254, -0.5      ],
258           [ 0.5      ,  0.8660254]])
259     >>> create_rotation_matrix(45)
260     array([[ 0.70710678, -0.70710678],
261           [ 0.70710678,  0.70710678]])
262     >>> create_rotation_matrix(np.pi / 3, degrees=False)
263     array([[ 0.5      , -0.8660254],
264           [ 0.8660254,  0.5      ]])
265
266     :param float angle: The angle to rotate anticlockwise by
267     :param bool degrees: Whether to interpret the angle as degrees (True) or radians (False)
268     :returns MatrixType: The resultant matrix
269     """
270     rad = np.deg2rad(angle) if degrees else angle
271     return np.array([
272         [np.cos(rad), -1 * np.sin(rad)],
273         [np.sin(rad), np.cos(rad)]
274     ])

```

A.7 gui/main_window.py

```

1  # lintrans - The linear transformation visualizer
2  # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4  # This program is licensed under GNU GPLv3, available here:
5  # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7  """This module provides the :class:`LintransMainWindow` class, which provides the main window for the GUI."""
8
9  from __future__ import annotations
10
11  import sys
12  import webbrowser
13  from copy import deepcopy
14  from typing import Type
15
16  import numpy as np
17  from numpy import linalg
18  from numpy.linalg import LinAlgError
19  from PyQt5 import QtWidgets
20  from PyQt5.QtCore import pyqtSlot, QThread
21  from PyQt5.QtGui import QKeySequence
22  from PyQt5.QtWidgets import QApplication, QHBoxLayout, QMainWindow, QMessageBox,
23  QShortcut, QSizePolicy, QSpacerItem, QVBoxLayout
24
25  from lintrans.matrices import MatrixWrapper
26  from lintrans.matrices.parse import validate_matrix_expression
27  from lintrans.typing_ import MatrixType
28  from . import dialogs
29  from .dialogs import DefineAsAnExpressionDialog, DefineDialog, DefineNumericallyDialog, DefineVisuallyDialog
30  from .dialogs.settings import DisplaySettingsDialog
31  from .plots import VisualizeTransformationWidget
32  from .settings import DisplaySettings
33  from .validate import MatrixExpressionValidator
34
35
36  class LintransMainWindow(QMainWindow):
37      """This class provides a main window for the GUI using the Qt framework.
38
39      This class should not be used directly, instead call :func:`lintrans.gui.main_window.main` to create the GUI.
40      """
41
42      def __init__(self):
43          """Create the main window object, and create and arrange every widget in it.
44
45          This doesn't show the window, it just constructs it. Use :func:`lintrans.gui.main_window.main` to show the
46      ↪ GUI.
47          """
48          super().__init__()
49
50          self.matrix_wrapper = MatrixWrapper()
51
52          self.setWindowTitle('lintrans')
53          self.setMinimumSize(1000, 750)
54
55          self.animating: bool = False
56          self.animating_sequence: bool = False
57
58          # === Create menubar
59
60          self.menubar = QtWidgets.QMenuBar(self)
61
62          self.menu_file = QtWidgets.QMenu(self.menubar)
63          self.menu_file.setTitle('&File')
64
65          self.menu_help = QtWidgets.QMenu(self.menubar)
66          self.menu_help.setTitle('&Help')
67
68          self.action_new = QtWidgets.QAction(self)
69          self.action_new.setText('&New')
70          self.action_new.setShortcut('Ctrl+N')

```



```

70     self.action_new.triggered.connect(lambda: print('new'))
71
72     self.action_open = QtWidgets.QAction(self)
73     self.action_open.setText('&Open')
74     self.action_open.setShortcut('Ctrl+O')
75     self.action_open.triggered.connect(lambda: print('open'))
76
77     self.action_save = QtWidgets.QAction(self)
78     self.action_save.setText('&Save')
79     self.action_save.setShortcut('Ctrl+S')
80     self.action_save.triggered.connect(lambda: print('save'))
81
82     self.action_save_as = QtWidgets.QAction(self)
83     self.action_save_as.setText('Save as...')
84     self.action_save_as.triggered.connect(lambda: print('save as'))
85
86     self.action_tutorial = QtWidgets.QAction(self)
87     self.action_tutorial.setText('&Tutorial')
88     self.action_tutorial.setShortcut('F1')
89     self.action_tutorial.triggered.connect(lambda: print('tutorial'))
90
91     self.action_docs = QtWidgets.QAction(self)
92     self.action_docs.setText('&Docs')
93     self.action_docs.triggered.connect(
94         lambda: webbrowser.open_new_tab('https://doctordalek1963.github.io/lintrans/docs/index.html')
95     )
96
97     self.action_about = QtWidgets.QAction(self)
98     self.action_about.setText('&About')
99     self.action_about.triggered.connect(lambda: dialogs.AboutDialog(self).open())
100
101     # TODO: Implement these actions and enable them
102     self.action_new.setEnabled(False)
103     self.action_open.setEnabled(False)
104     self.action_save.setEnabled(False)
105     self.action_save_as.setEnabled(False)
106     self.action_tutorial.setEnabled(False)
107
108     self.menu_file.addAction(self.action_new)
109     self.menu_file.addAction(self.action_open)
110     self.menu_file.addSeparator()
111     self.menu_file.addAction(self.action_save)
112     self.menu_file.addAction(self.action_save_as)
113
114     self.menu_help.addAction(self.action_tutorial)
115     self.menu_help.addAction(self.action_docs)
116     self.menu_help.addSeparator()
117     self.menu_help.addAction(self.action_about)
118
119     self.menubar.addAction(self.menu_file.menuAction())
120     self.menubar.addAction(self.menu_help.menuAction())
121
122     self.setMenuBar(self.menubar)
123
124     # === Create widgets
125
126     # Left layout: the plot and input box
127
128     self.plot = VisualizeTransformationWidget(self, display_settings=DisplaySettings())
129
130     self.lineedit_expression_box = QtWidgets.QLineEdit(self)
131     self.lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
132     self.lineedit_expression_box.setValidator(MatrixExpressionValidator(self))
133     self.lineedit_expression_box.textChanged.connect(self.update_render_buttons)
134
135     # Right layout: all the buttons
136
137     # Misc buttons
138
139     self.button_create_polygon = QtWidgets.QPushButton(self)
140     self.button_create_polygon.setText('Create polygon')
141     # self.button_create_polygon.clicked.connect(self.create_polygon)
142     self.button_create_polygon.setToolTip('Define a new polygon to view the transformation of')

```

```

143
144     # TODO: Implement this and enable button
145     self.button_create_polygon.setEnabled(False)
146
147     self.button_change_display_settings = QtWidgets.QPushButton(self)
148     self.button_change_display_settings.setText('Change\ndisplay settings')
149     self.button_change_display_settings.clicked.connect(self.dialog_change_display_settings)
150     self.button_change_display_settings.setToolTip(
151         "Change which things are rendered and how they're rendered<br><b>(Ctrl + D)</b>"
152     )
153     QShortcut(QKeySequence('Ctrl+D'), self).activated.connect(self.button_change_display_settings.click)
154
155     self.button_reset_zoom = QtWidgets.QPushButton(self)
156     self.button_reset_zoom.setText('Reset zoom')
157     self.button_reset_zoom.clicked.connect(self.reset_zoom)
158     self.button_reset_zoom.setToolTip('Reset the zoom level back to normal<br><b>(Ctrl + Shift + R)</b>')
159     QShortcut(QKeySequence('Ctrl+Shift+R'), self).activated.connect(self.button_reset_zoom.click)
160
161     # Define new matrix buttons and their groupbox
162
163     self.button_define_visually = QtWidgets.QPushButton(self)
164     self.button_define_visually.setText('Visually')
165     self.button_define_visually.setToolTip('Drag the basis vectors<br><b>(Alt + 1)</b>')
166     self.button_define_visually.clicked.connect(lambda: self.dialog_define_matrix(DefineVisuallyDialog))
167     QShortcut(QKeySequence('Alt+1'), self).activated.connect(self.button_define_visually.click)
168
169     self.button_define_numerically = QtWidgets.QPushButton(self)
170     self.button_define_numerically.setText('Numerically')
171     self.button_define_numerically.setToolTip('Define a matrix just with numbers<br><b>(Alt + 2)</b>')
172     self.button_define_numerically.clicked.connect(lambda: self.dialog_define_matrix(DefineNumericallyDialog))
173     QShortcut(QKeySequence('Alt+2'), self).activated.connect(self.button_define_numerically.click)
174
175     self.button_define_as_expression = QtWidgets.QPushButton(self)
176     self.button_define_as_expression.setText('As an expression')
177     self.button_define_as_expression.setToolTip('Define a matrix in terms of other matrices<br><b>(Alt +
178     ↩ 3)</b>')
179     self.button_define_as_expression.clicked.connect(lambda:
180     ↩ self.dialog_define_matrix(DefineAsAnExpressionDialog))
181     QShortcut(QKeySequence('Alt+3'), self).activated.connect(self.button_define_as_expression.click)
182
183     self.vlay_define_new_matrix = QVBoxLayout()
184     self.vlay_define_new_matrix.setSpacing(20)
185     self.vlay_define_new_matrix.addWidget(self.button_define_visually)
186     self.vlay_define_new_matrix.addWidget(self.button_define_numerically)
187     self.vlay_define_new_matrix.addWidget(self.button_define_as_expression)
188
189     self.groupbox_define_new_matrix = QtWidgets.QGroupBox('Define a new matrix', self)
190     self.groupbox_define_new_matrix.setLayout(self.vlay_define_new_matrix)
191
192     # Render buttons
193
194     self.button_reset = QtWidgets.QPushButton(self)
195     self.button_reset.setText('Reset')
196     self.button_reset.clicked.connect(self.reset_transformation)
197     self.button_reset.setToolTip('Reset the visualized transformation back to the identity<br><b>(Ctrl +
198     ↩ R)</b>')
199     QShortcut(QKeySequence('Ctrl+R'), self).activated.connect(self.button_reset.click)
200
201     self.button_render = QtWidgets.QPushButton(self)
202     self.button_render.setText('Render')
203     self.button_render.setEnabled(False)
204     self.button_render.clicked.connect(self.render_expression)
205     self.button_render.setToolTip('Render the expression<br><b>(Ctrl + Enter)</b>')
206     QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self.button_render.click)
207
208     self.button_animate = QtWidgets.QPushButton(self)
209     self.button_animate.setText('Animate')
210     self.button_animate.setEnabled(False)
211     self.button_animate.clicked.connect(self.animate_expression)
212     self.button_animate.setToolTip('Animate the expression<br><b>(Ctrl + Shift + Enter)</b>')
213     QShortcut(QKeySequence('Ctrl+Shift+Return'), self).activated.connect(self.button_animate.click)
214
215     # == Arrange widgets

```

```

213
214     self.vlay_left = QVBoxLayout()
215     self.vlay_left.addWidget(self.plot)
216     self.vlay_left.addWidget(self.linedit_expression_box)
217
218     self.vlay_misc_buttons = QVBoxLayout()
219     self.vlay_misc_buttons.setSpacing(20)
220     self.vlay_misc_buttons.addWidget(self.button_create_polygon)
221     self.vlay_misc_buttons.addWidget(self.button_change_display_settings)
222     self.vlay_misc_buttons.addWidget(self.button_reset_zoom)
223
224     self.vlay_render = QVBoxLayout()
225     self.vlay_render.setSpacing(20)
226     self.vlay_render.addWidget(self.button_reset)
227     self.vlay_render.addWidget(self.button_animate)
228     self.vlay_render.addWidget(self.button_render)
229
230     self.vlay_right = QVBoxLayout()
231     self.vlay_right.setSpacing(50)
232     self.vlay_right.addLayout(self.vlay_misc_buttons)
233     self.vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
234     self.vlay_right.addWidget(self.groupbox_define_new_matrix)
235     self.vlay_right.addItem(QSpacerItem(100, 2, hPolicy=QSizePolicy.Minimum, vPolicy=QSizePolicy.Expanding))
236     self.vlay_right.addLayout(self.vlay_render)
237
238     self.hlay_all = QHBoxLayout()
239     self.hlay_all.setSpacing(15)
240     self.hlay_all.addLayout(self.vlay_left)
241     self.hlay_all.addLayout(self.vlay_right)
242
243     self.central_widget = QtWidgets.QWidget()
244     self.central_widget.setLayout(self.hlay_all)
245     self.central_widget.setContentsMargins(10, 10, 10, 10)
246
247     self.setCentralWidget(self.central_widget)
248
249 def update_render_buttons(self) -> None:
250     """Enable or disable the render and animate buttons according to whether the matrix expression is valid."""
251     text = self.linedit_expression_box.text()
252
253     # Let's say that the user defines a non-singular matrix A, then defines B as A^-1
254     # If they then redefine A and make it singular, then we get a LinAlgError when
255     # trying to evaluate an expression with B in it
256     # To fix this, we just do naive validation rather than aware validation
257     if ',' in text:
258         self.button_render.setEnabled(False)
259
260         try:
261             valid = all(self.matrix_wrapper.is_valid_expression(x) for x in text.split(','))
262         except LinAlgError:
263             valid = all(validate_matrix_expression(x) for x in text.split(','))
264
265         self.button_animate.setEnabled(valid)
266
267     else:
268         try:
269             valid = self.matrix_wrapper.is_valid_expression(text)
270         except LinAlgError:
271             valid = validate_matrix_expression(text)
272
273         self.button_render.setEnabled(valid)
274         self.button_animate.setEnabled(valid)
275
276 @pyqtSlot()
277 def reset_zoom(self) -> None:
278     """Reset the zoom level back to normal."""
279     self.plot.grid_spacing = self.plot.default_grid_spacing
280     self.plot.update()
281
282 @pyqtSlot()
283 def reset_transformation(self) -> None:
284     """Reset the visualized transformation back to the identity."""
285     self.plot.visualize_matrix_transformation(self.matrix_wrapper['I'])

```

```

286         self.animating = False
287         self.animating_sequence = False
288         self.plot.update()
289
290     @pyqtSlot()
291     def render_expression(self) -> None:
292         """Render the transformation given by the expression in the input box."""
293         try:
294             matrix = self.matrix_wrapper.evaluate_expression(self.lineedit_expression_box.text())
295
296         except LinAlgError:
297             self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
298             return
299
300         if self.is_matrix_too_big(matrix):
301             self.show_error_message('Matrix too big', "This matrix doesn't fit on the canvas")
302             return
303
304         self.plot.visualize_matrix_transformation(matrix)
305         self.plot.update()
306
307     @pyqtSlot()
308     def animate_expression(self) -> None:
309         """Animate from the current matrix to the matrix in the expression box."""
310         self.button_render.setEnabled(False)
311         self.button_animate.setEnabled(False)
312
313         matrix_start: MatrixType = np.array([
314             [self.plot.point_i[0], self.plot.point_j[0]],
315             [self.plot.point_i[1], self.plot.point_j[1]]
316         ])
317
318         text = self.lineedit_expression_box.text()
319
320         # If there's commas in the expression, then we want to animate each part at a time
321         if ',' in text:
322             current_matrix = matrix_start
323             self.animating_sequence = True
324
325             # For each expression in the list, right multiply it by the current matrix,
326             # and animate from the current matrix to that new matrix
327             for expr in text.split(',')[:-1]:
328                 try:
329                     new_matrix = self.matrix_wrapper.evaluate_expression(expr) @ current_matrix
330                 except LinAlgError:
331                     self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
332                     return
333
334                 if not self.animating_sequence:
335                     break
336
337                 self.animate_between_matrices(current_matrix, new_matrix)
338                 current_matrix = new_matrix
339
340                 # Here we just redraw and allow for other events to be handled while we pause
341                 self.plot.update()
342                 QApplication.processEvents()
343                 QThread.msleep(self.plot.display_settings.animation_pause_length)
344
345             self.animating_sequence = False
346
347         # If there's no commas, then just animate directly from the start to the target
348         else:
349             # Get the target matrix and it's determinant
350             try:
351                 matrix_target = self.matrix_wrapper.evaluate_expression(text)
352
353             except LinAlgError:
354                 self.show_error_message('Singular matrix', 'Cannot take inverse of singular matrix')
355                 return
356
357             # The concept of applicative animation is explained in /gui/settings.py
358             if self.plot.display_settings.applicative_animation:

```

```

359         matrix_target = matrix_target @ matrix_start
360
361         # If we want a transitional animation and we're animating the same matrix, then restart the animation
362         # We use this check rather than equality because of small floating point errors
363         elif (abs(matrix_start - matrix_target) < 1e-12).all():
364             matrix_start = self.matrix_wrapper['I']
365
366             # We pause here for 200 ms to make the animation look a bit nicer
367             self.plot.visualize_matrix_transformation(matrix_start)
368             self.plot.update()
369             QApplication.processEvents()
370             QThread.sleep(200)
371
372         self.animate_between_matrices(matrix_start, matrix_target)
373
374     self.update_render_buttons()
375
376 def animate_between_matrices(self, matrix_start: MatrixType, matrix_target: MatrixType, steps: int = 100) ->
↪ None:
377     """Animate from the start matrix to the target matrix."""
378     det_target = linalg.det(matrix_target)
379     det_start = linalg.det(matrix_start)
380
381     self.animating = True
382
383     for i in range(0, steps + 1):
384         if not self.animating:
385             break
386
387         # This proportion is how far we are through the loop
388         proportion = i / steps
389
390         # matrix_a is the start matrix plus some part of the target, scaled by the proportion
391         # If we just used matrix_a, then things would animate, but the determinants would be weird
392         matrix_a = matrix_start + proportion * (matrix_target - matrix_start)
393
394         if self.plot.display_settings.smoothen_determinant and det_start * det_target > 0:
395             # To fix the determinant problem, we get the determinant of matrix_a and use it to normalize
396             det_a = linalg.det(matrix_a)
397
398             # For a 2x2 matrix A and a scalar c, we know that det(cA) = c^2 det(A)
399             # We want B = cA such that det(B) = det(S), where S is the start matrix,
400             # so then we can scale it with the animation, so we get
401             # det(cA) = c^2 det(A) = det(S) => c = sqrt(abs(det(S) / det(A)))
402             # Then we scale A to get the determinant we want, and call that matrix_b
403             if det_a == 0:
404                 c = 0
405             else:
406                 c = np.sqrt(abs(det_start / det_a))
407
408             matrix_b = c * matrix_a
409             det_b = linalg.det(matrix_b)
410
411             # matrix_to_render is the final matrix that we then render for this frame
412             # It's B, but we scale it over time to have the target determinant
413
414             # We want some C = dB such that det(C) is some target determinant T
415             # det(dB) = d^2 det(B) = T => d = sqrt(abs(T / det(B)))
416
417             # We're also subtracting 1 and multiplying by the proportion and then adding one
418             # This just scales the determinant along with the animation
419
420             # That is all of course, if we can do that
421             # We'll crash if we try to do this with det(B) == 0
422             if det_b != 0:
423                 scalar = 1 + proportion * (np.sqrt(abs(det_target / det_b)) - 1)
424                 matrix_to_render = scalar * matrix_b
425
426             else:
427                 matrix_to_render = matrix_a
428
429         else:
430             matrix_to_render = matrix_a

```

```

431
432         if self.is_matrix_too_big(matrix_to_render):
433             self.show_error_message('Matrix too big', "This matrix doesn't fit on the canvas")
434             return
435
436         self.plot.visualize_matrix_transformation(matrix_to_render)
437
438         # We schedule the plot to be updated, tell the event loop to
439         # process events, and asynchronously sleep for 10ms
440         # This allows for other events to be processed while animating, like zooming in and out
441         self.plot.update()
442         QApplication.processEvents()
443         QThread.msleep(1000 // steps)
444
445     self.animating = False
446
447 @pyqtSlot(DefineDialog)
448 def dialog_define_matrix(self, dialog_class: Type[DefineDialog]) -> None:
449     """Open a generic definition dialog to define a new matrix.
450
451     The class for the desired dialog is passed as an argument. We create an
452     instance of this class and the dialog is opened asynchronously and modally
453     (meaning it blocks interaction with the main window) with the proper method
454     connected to the :meth:`QDialog.accepted` signal.
455
456     .. note:: ``dialog_class`` must subclass :class:`lintrans.gui.dialogs.define_new_matrix.DefineDialog`.
457
458     :param dialog_class: The dialog class to instantiate
459     :type dialog_class: Type[lintrans.gui.dialogs.define_new_matrix.DefineDialog]
460     """
461     # We create a dialog with a deepcopy of the current matrix_wrapper
462     # This avoids the dialog mutating this one
463     dialog: DefineDialog
464
465     if dialog_class == DefineVisuallyDialog:
466         dialog = DefineVisuallyDialog(
467             self,
468             matrix_wrapper=deepcopy(self.matrix_wrapper),
469             display_settings=self.plot.display_settings
470         )
471     else:
472         dialog = dialog_class(self, matrix_wrapper=deepcopy(self.matrix_wrapper))
473
474     # .open() is asynchronous and doesn't spawn a new event loop, but the dialog is still modal (blocking)
475     dialog.open()
476
477     # So we have to use the accepted signal to call a method when the user accepts the dialog
478     dialog.accepted.connect(self.assign_matrix_wrapper)
479
480 @pyqtSlot()
481 def assign_matrix_wrapper(self) -> None:
482     """Assign a new value to ``self.matrix_wrapper`` and give the expression box focus."""
483     self.matrix_wrapper = self.sender().matrix_wrapper
484     self.linedit_expression_box.setFocus()
485     self.update_render_buttons()
486
487 @pyqtSlot()
488 def dialog_change_display_settings(self) -> None:
489     """Open the dialog to change the display settings."""
490     dialog = DisplaySettingsDialog(self, display_settings=self.plot.display_settings)
491     dialog.open()
492     dialog.accepted.connect(lambda: self.assign_display_settings(dialog.display_settings))
493
494 @pyqtSlot(DisplaySettings)
495 def assign_display_settings(self, display_settings: DisplaySettings) -> None:
496     """Assign a new value to ``self.plot.display_settings`` and give the expression box focus."""
497     self.plot.display_settings = display_settings
498     self.plot.update()
499     self.linedit_expression_box.setFocus()
500     self.update_render_buttons()
501
502 def show_error_message(self, title: str, text: str, info: str | None = None) -> None:
503     """Show an error message in a dialog box.
```

```

504
505     :param str title: The window title of the dialog box
506     :param str text: The simple error message
507     :param info: The more informative error message
508     :type info: Optional[str]
509     """
510     dialog = QMessageBox(self)
511     dialog.setIcon(QMessageBox.Critical)
512     dialog.setWindowTitle(title)
513     dialog.setText(text)
514
515     if info is not None:
516         dialog.setInformativeText(info)
517
518     dialog.open()
519
520     # This is `finished` rather than `accepted` because we want to update the buttons no matter what
521     dialog.finished.connect(self.update_render_buttons)
522
523     def is_matrix_too_big(self, matrix: MatrixType) -> bool:
524         """Check if the given matrix will actually fit onto the canvas.
525
526         Convert the elements of the matrix to canvas coords and make sure they fit within Qt's 32-bit integer limit.
527
528         :param MatrixType matrix: The matrix to check
529         :returns bool: Whether the matrix fits on the canvas
530         """
531         coords: list[tuple[int, int]] = [self.plot.canvas_coords(*vector) for vector in matrix.T]
532
533         for x, y in coords:
534             if not (-2147483648 <= x <= 2147483647 and -2147483648 <= y <= 2147483647):
535                 return True
536
537         return False
538
539
540     def main(args: list[str]) -> None:
541         """Run the GUI by creating and showing an instance of :class:`LintransMainWindow`.
542
543         :param list[str] args: The args to pass to :class:`QApplication`
544         """
545         app = QApplication(args)
546         window = LintransMainWindow()
547         window.show()
548         sys.exit(app.exec_())

```

A.8 gui/validate.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 simple module provides a :class:`MatrixExpressionValidator` class to validate matrix expression input."""
8
9  from __future__ import annotations
10
11  import re
12
13  from PyQt5.QtGui import QValidator
14
15  from lintrans.matrices import parse
16
17
18  class MatrixExpressionValidator(QValidator):
19     """This class validates matrix expressions in a Qt input box."""
20
21     def validate(self, text: str, pos: int) -> tuple[QValidator.State, str, int]:
22         """Validate the given text according to the rules defined in the :mod:`lintrans.matrices` module."""

```

```

23     clean_text = re.sub(parse.NAIVE_CHARACTER_CLASS[:-1] + ',]', '', text)
24
25     if clean_text == '':
26         if parse.validate_matrix_expression(clean_text):
27             return QValidator.Acceptable, text, pos
28         else:
29             return QValidator.Intermediate, text, pos
30
31     return QValidator.Invalid, text, pos

```

A.9 gui/settings.py

```

1  # lintrans - The linear transformation visualizer
2  # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4  # This program is licensed under GNU GPLv3, available here:
5  # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7  """This module contains the :class:`DisplaySettings` class, which holds configuration for display."""
8
9  from __future__ import annotations
10
11  from dataclasses import dataclass
12
13
14  @dataclass
15  class DisplaySettings:
16      """This class simply holds some attributes to configure display."""
17
18      # === Basic stuff
19
20      draw_background_grid: bool = True
21      """This controls whether we want to draw the background grid.
22
23      The background axes will always be drawn. This makes it easy to identify the center of the space.
24      """
25
26      draw_transformed_grid: bool = True
27      """This controls whether we want to draw the transformed grid. Vectors are handled separately."""
28
29      draw_basis_vectors: bool = True
30      """This controls whether we want to draw the transformed basis vectors."""
31
32      # === Animations
33
34      smoothen_determinant: bool = True
35      """This controls whether we want the determinant to change smoothly during the animation.
36
37      .. note::
38          Even if this is True, it will be ignored if we're animating from a positive det matrix to
39          a negative det matrix, or vice versa, because if we try to smoothly animate that determinant,
40          things blow up and the app often crashes.
41      """
42
43      applicative_animation: bool = True
44      """There are two types of simple animation, transitional and applicative.
45
46      Let ``C`` be the matrix representing the currently displayed transformation, and let ``T`` be the target matrix.
47      Transitional animation means that we animate directly from ``C`` from ``T``,
48      and applicative animation means that we animate from ``C`` to ``TC``, so we apply ``T`` to ``C``.
49      """
50
51      animation_pause_length: int = 400
52      """This is the number of milliseconds that we wait between animations when using comma syntax."""
53
54      # === Matrix info
55
56      draw_determinant_parallelogram: bool = False
57      """This controls whether or not we should shade the parallelogram representing the determinant of the matrix."""
58

```



```

59     show_determinant_value: bool = True
60     """This controls whether we should write the text value of the determinant inside the parallelogram.
61
62     The text only gets draw if :attr:`draw_determinant_parallelogram` is also True.
63     """
64
65     draw_eigenvectors: bool = False
66     """This controls whether we should draw the eigenvectors of the transformation."""
67
68     draw_eigenlines: bool = False
69     """This controls whether we should draw the eigenlines of the transformation."""

```

A.10 gui/__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 the main GUI and associated dialogs for visualization."""
8
9  from . import dialogs, plots, settings, validate
10 from .main_window import main
11
12 __all__ = ['dialogs', 'main', 'plots', 'settings', 'validate']

```

A.11 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:`DefinedDialog` class and subclasses, allowing definition of new
  ↪ matrices."""
8
9  from __future__ import annotations
10
11 import abc
12
13 from numpy import array
14 from PyQt5 import QtWidgets
15 from PyQt5.QtCore import pyqtSlot
16 from PyQt5.QtGui import QDoubleValidator, QKeySequence
17 from PyQt5.QtWidgets import QGridLayout, QHBoxLayout, QShortcut, QSizePolicy, QSpacerItem, QVBoxLayout
18
19 from lintrans.gui.dialogs.misc import FixedSizeDialog
20 from lintrans.gui.plots import DefineVisuallyWidget
21 from lintrans.gui.settings import DisplaySettings
22 from lintrans.gui.validate import MatrixExpressionValidator
23 from lintrans.matrices import MatrixWrapper
24 from lintrans.typing_ import MatrixType
25
26 ALPHABET_NO_I = 'ABCDEFGHJKLMNPOQRSTUVWXYZ'
27
28
29 def is_valid_float(string: str) -> bool:
30     """Check if the string is a valid float (or anything that can be cast to a float, such as an int).
31
32     This function simply checks that ``float(string)`` doesn't raise an error.
33
34     .. note:: An empty string is not a valid float, so will return False.
35
36     :param str string: The string to check
37     :returns bool: Whether the string is a valid float
38     """

```

```

39     try:
40         float(string)
41         return True
42     except ValueError:
43         return False
44
45
46 def round_float(num: float, precision: int = 5) -> str:
47     """Round a floating point number to a given number of decimal places for pretty printing.
48
49     :param float num: The number to round
50     :param int precision: The number of decimal places to round to
51     :returns str: The rounded number for pretty printing
52     """
53     # Round to ``precision`` number of decimal places
54     string = str(round(num, precision))
55
56     # Cut off the potential final zero
57     if string.endswith('.0'):
58         return string[:-2]
59
60     elif 'e' in string: # Scientific notation
61         split = string.split('e')
62         # The leading 0 only happens when the exponent is negative, so we know there'll be a minus sign
63         return split[0] + 'e-' + split[1][1:].lstrip('0')
64
65     else:
66         return string
67
68
69 class DefineDialog(FixedSizeDialog):
70     """An abstract superclass for definitions dialogs.
71
72     .. warning:: This class should never be directly instantiated, only subclassed.
73
74     .. note::
75         I would make this class have ``metaclass=abc.ABCMeta``, but I can't because it subclasses :class:`QDialog`,
76         and every superclass of a class must have the same metaclass, and :class:`QDialog` is not an abstract class.
77     """
78
79     def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
80         """Create the widgets and layout of the dialog.
81
82         .. note:: ``*args`` and ``**kwargs`` are passed to the super constructor (:class:`QDialog`).
83
84         :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
85         """
86         super().__init__(*args, **kwargs)
87
88         self.matrix_wrapper = matrix_wrapper
89         self.setWindowTitle('Define a matrix')
90
91         # === Create the widgets
92
93         self.button_confirm = QtWidgets.QPushButton(self)
94         self.button_confirm.setText('Confirm')
95         self.button_confirm.setEnabled(False)
96         self.button_confirm.clicked.connect(self.confirm_matrix)
97         self.button_confirm.setToolTip('Confirm this as the new matrix<br><b>(Ctrl + Enter)</b>')
98         QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self.button_confirm.click)
99
100         self.button_cancel = QtWidgets.QPushButton(self)
101         self.button_cancel.setText('Cancel')
102         self.button_cancel.clicked.connect(self.reject)
103         self.button_cancel.setToolTip('Cancel this definition<br><b>(Escape)</b>')
104
105         self.label_equals = QtWidgets.QLabel()
106         self.label_equals.setText('=')
107
108         self.combobox_letter = QtWidgets.QComboBox(self)
109
110         for letter in ALPHABET_NO_I:
111             self.combobox_letter.addItem(letter)

```

```

112         self.combobox_letter.activated.connect(self.load_matrix)
113
114     # === Arrange the widgets
115
116     self.setContentsMargins(10, 10, 10, 10)
117
118     self.hlay_buttons = QHBoxLayout()
119     self.hlay_buttons.setSpacing(20)
120     self.hlay_buttons.addItem(QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum))
121     self.hlay_buttons.addWidget(self.button_cancel)
122     self.hlay_buttons.addWidget(self.button_confirm)
123
124     self.hlay_definition = QHBoxLayout()
125     self.hlay_definition.setSpacing(20)
126     self.hlay_definition.addWidget(self.combobox_letter)
127     self.hlay_definition.addWidget(self.label_equals)
128
129     self.vlay_all = QVBoxLayout()
130     self.vlay_all.setSpacing(20)
131
132     self.setLayout(self.vlay_all)
133
134
135 @property
136 def selected_letter(self) -> str:
137     """Return the letter currently selected in the combo box."""
138     return str(self.combobox_letter.currentText())
139
140 @abc.abstractmethod
141 @pyqtSlot()
142 def update_confirm_button(self) -> None:
143     """Enable the confirm button if it should be enabled, else, disable it."""
144
145 @pyqtSlot(int)
146 def load_matrix(self, index: int) -> None:
147     """Load the selected matrix into the dialog.
148
149     This method is optionally able to be overridden. If it is not overridden,
150     then no matrix is loaded when selecting a name.
151
152     We have this method in the superclass so that we can define it as the slot
153     for the :meth:`QComboBox.activated` signal in this constructor, rather than
154     having to define that in the constructor of every subclass.
155     """
156
157 @abc.abstractmethod
158 @pyqtSlot()
159 def confirm_matrix(self) -> None:
160     """Confirm the inputted matrix and assign it.
161
162     .. note:: When subclassing, this method should mutate ``self.matrix_wrapper`` and then call
163     ↪ ``self.accept()``.
164     """
165
166 class DefineVisuallyDialog(DefineDialog):
167     """The dialog class that allows the user to define a matrix visually."""
168
169     def __init__(self, *args, matrix_wrapper: MatrixWrapper, display_settings: DisplaySettings, **kwargs):
170         """Create the widgets and layout of the dialog.
171
172         :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
173         """
174         super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
175
176         self.setMinimumSize(700, 550)
177
178     # === Create the widgets
179
180     self.plot = DefineVisuallyWidget(self, display_settings=display_settings)
181
182     # === Arrange the widgets
183

```

```

184         self.hlay_definition.addWidget(self.plot)
185         self.hlay_definition.setStretchFactor(self.plot, 1)
186
187         self.vlay_all.addLayout(self.hlay_definition)
188         self.vlay_all.addLayout(self.hlay_buttons)
189
190         # We load the default matrix A into the plot
191         self.load_matrix(0)
192
193         # We also enable the confirm button, because any visually defined matrix is valid
194         self.button_confirm.setEnabled(True)
195
196     @pyqtSlot()
197     def update_confirm_button(self) -> None:
198         """Enable the confirm button.
199
200         .. note::
201             The confirm button is always enabled in this dialog and this method is never actually used,
202             so it's got an empty body. It's only here because we need to implement the abstract method.
203         """
204
205     @pyqtSlot(int)
206     def load_matrix(self, index: int) -> None:
207         """Show the selected matrix on the plot. If the matrix is None, show the identity."""
208         matrix = self.matrix_wrapper[self.selected_letter]
209
210         if matrix is None:
211             matrix = self.matrix_wrapper['I']
212
213         self.plot.visualize_matrix_transformation(matrix)
214         self.plot.update()
215
216     @pyqtSlot()
217     def confirm_matrix(self) -> None:
218         """Confirm the matrix that's been defined visually."""
219         matrix: MatrixType = array([
220             [self.plot.point_i[0], self.plot.point_j[0]],
221             [self.plot.point_i[1], self.plot.point_j[1]]
222         ])
223
224         self.matrix_wrapper[self.selected_letter] = matrix
225         self.accept()
226
227
228 class DefineNumericallyDialog(DefinedDialog):
229     """The dialog class that allows the user to define a new matrix numerically."""
230
231     def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
232         """Create the widgets and layout of the dialog.
233
234         :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
235         """
236         super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
237
238         # === Create the widgets
239
240         # tl = top left, br = bottom right, etc.
241         self.element_tl = QtWidgets.QLineEdit(self)
242         self.element_tl.textChanged.connect(self.update_confirm_button)
243         self.element_tl.setValidator(QDoubleValidator())
244
245         self.element_tr = QtWidgets.QLineEdit(self)
246         self.element_tr.textChanged.connect(self.update_confirm_button)
247         self.element_tr.setValidator(QDoubleValidator())
248
249         self.element_bl = QtWidgets.QLineEdit(self)
250         self.element_bl.textChanged.connect(self.update_confirm_button)
251         self.element_bl.setValidator(QDoubleValidator())
252
253         self.element_br = QtWidgets.QLineEdit(self)
254         self.element_br.textChanged.connect(self.update_confirm_button)
255         self.element_br.setValidator(QDoubleValidator())
256

```

```

257         self.matrix_elements = (self.element_tl, self.element_tr, self.element_bl, self.element_br)
258
259         # === Arrange the widgets
260
261         self.grid_matrix = QGridLayout()
262         self.grid_matrix.setSpacing(20)
263         self.grid_matrix.addWidget(self.element_tl, 0, 0)
264         self.grid_matrix.addWidget(self.element_tr, 0, 1)
265         self.grid_matrix.addWidget(self.element_bl, 1, 0)
266         self.grid_matrix.addWidget(self.element_br, 1, 1)
267
268         self.hlay_definition.addLayout(self.grid_matrix)
269
270         self.vlay_all.addLayout(self.hlay_definition)
271         self.vlay_all.addLayout(self.hlay_buttons)
272
273         # We load the default matrix A into the boxes
274         self.load_matrix(0)
275
276         self.element_tl.setFocus()
277
278     @pyqtSlot()
279     def update_confirm_button(self) -> None:
280         """Enable the confirm button if there are valid floats in every box."""
281         for elem in self.matrix_elements:
282             if not is_valid_float(elem.text()):
283                 # If they're not all numbers, then we can't confirm it
284                 self.button_confirm.setEnabled(False)
285                 return
286
287         # If we didn't find anything invalid
288         self.button_confirm.setEnabled(True)
289
290     @pyqtSlot(int)
291     def load_matrix(self, index: int) -> None:
292         """If the selected matrix is defined, load its values into the boxes."""
293         matrix = self.matrix_wrapper[self.selected_letter]
294
295         if matrix is None:
296             for elem in self.matrix_elements:
297                 elem.setText('')
298
299         else:
300             self.element_tl.setText(round_float(matrix[0][0]))
301             self.element_tr.setText(round_float(matrix[0][1]))
302             self.element_bl.setText(round_float(matrix[1][0]))
303             self.element_br.setText(round_float(matrix[1][1]))
304
305         self.update_confirm_button()
306
307     @pyqtSlot()
308     def confirm_matrix(self) -> None:
309         """Confirm the matrix in the boxes and assign it to the name in the combo box."""
310         matrix: MatrixType = array([
311             [float(self.element_tl.text()), float(self.element_tr.text())],
312             [float(self.element_bl.text()), float(self.element_br.text())]
313         ])
314
315         self.matrix_wrapper[self.selected_letter] = matrix
316         self.accept()
317
318
319 class DefineAsAnExpressionDialog(DefineDialog):
320     """The dialog class that allows the user to define a matrix as an expression of other matrices."""
321
322     def __init__(self, *args, matrix_wrapper: MatrixWrapper, **kwargs):
323         """Create the widgets and layout of the dialog.
324
325         :param MatrixWrapper matrix_wrapper: The MatrixWrapper that this dialog will mutate
326         """
327         super().__init__(*args, matrix_wrapper=matrix_wrapper, **kwargs)
328
329         self.setMinimumWidth(450)

```

```

330
331     # === Create the widgets
332
333     self.lineedit_expression_box = QtWidgets.QLineEdit(self)
334     self.lineedit_expression_box.setPlaceholderText('Enter matrix expression...')
335     self.lineedit_expression_box.textChanged.connect(self.update_confirm_button)
336     self.lineedit_expression_box.setValidator(MatrixExpressionValidator())
337
338     # === Arrange the widgets
339
340     self.hlay_definition.addWidget(self.lineedit_expression_box)
341
342     self.vlay_all.addLayout(self.hlay_definition)
343     self.vlay_all.addLayout(self.hlay_buttons)
344
345     # Load the matrix if it's defined as an expression
346     self.load_matrix(0)
347
348     self.lineedit_expression_box.setFocus()
349
350     @pyqtSlot()
351     def update_confirm_button(self) -> None:
352         """Enable the confirm button if the matrix expression is valid in the wrapper."""
353         text = self.lineedit_expression_box.text()
354         valid_expression = self.matrix_wrapper.is_valid_expression(text)
355
356         self.button_confirm.setEnabled(valid_expression and self.selected_letter not in text)
357
358     @pyqtSlot(int)
359     def load_matrix(self, index: int) -> None:
360         """If the selected matrix is defined an expression, load that expression into the box."""
361         if (expr := self.matrix_wrapper.get_expression(self.selected_letter)) is not None:
362             self.lineedit_expression_box.setText(expr)
363         else:
364             self.lineedit_expression_box.setText('')
365
366     @pyqtSlot()
367     def confirm_matrix(self) -> None:
368         """Evaluate the matrix expression and assign its value to the name in the combo box."""
369         self.matrix_wrapper[self.selected_letter] = self.lineedit_expression_box.text()
370         self.accept()

```

A.12 gui/dialogs/misc.py

```

1  # lintrans - The linear transformation visualizer
2  # Copyright (C) 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 platform
12
13  from PyQt5 import QtWidgets
14  from PyQt5.QtCore import Qt
15  from PyQt5.QtWidgets import QDialog, QVBoxLayout
16
17  import lintrans
18
19
20  class FixedSizeDialog(QDialog):
21     """A simple superclass to create modal dialog boxes with fixed size.
22
23     We override the :meth:`open` method to set the fixed size as soon as the dialog is opened modally.
24     """
25
26     def open(self) -> None:

```

```

27         """Override :meth:`QDialog.open` to set the dialog to a fixed size."""
28         super().open()
29         self.setFixedSize(self.size())
30
31
32 class AboutDialog(FixedSizeDialog):
33     """A simple dialog class to display information about the app to the user.
34
35     It only has an :meth:`__init__` method because it only has label widgets, so no other methods are necessary
36     here.
37     """
38
39     def __init__(self, *args, **kwargs):
40         """Create an :class:`AboutDialog` object with all the label widgets."""
41         super().__init__(*args, **kwargs)
42
43         self.setWindowTitle('About lintrans')
44
45         # === Create the widgets
46
47         label_title = QtWidgets.QLabel(self)
48         label_title.setText(f'lintrans (version {lintrans.__version__})')
49         label_title.setAlignment(Qt.AlignCenter)
50
51         font_title = label_title.font()
52         font_title.setPointSize(font_title.pointSize() * 2)
53         label_title.setFont(font_title)
54
55         label_version_info = QtWidgets.QLabel(self)
56         label_version_info.setText(
57             f'With Python version {platform.python_version()}\n'
58             f'Running on {platform.platform()}'
59         )
60         label_version_info.setAlignment(Qt.AlignCenter)
61
62         label_info = QtWidgets.QLabel(self)
63         label_info.setText(
64             'lintrans is a program designed to help visualise<br>'
65             '2D linear transformations represented with matrices.<br><br>'
66             'It's designed for teachers and students and any feedback<br>'
67             'is greatly appreciated at <a href="https://github.com/DoctorDalek1963/lintrans" '
68             'style="color: black;">my GitHub page</a><br>or via email '
69             '(<a href="mailto:dyson.dyson@icloud.com" style="color: black;">dyson.dyson@icloud.com</a>).'
70         )
71         label_info.setAlignment(Qt.AlignCenter)
72         label_info.setTextFormat(Qt.RichText)
73         label_info.setOpenExternalLinks(True)
74
75         label_copyright = QtWidgets.QLabel(self)
76         label_copyright.setText(
77             'This program is free software.<br>Copyright 2021-2022 D. Dyson (DoctorDalek1963).<br>'
78             'This program is licensed under GPLv3, which can be found '
79             '<a href="https://www.gnu.org/licenses/gpl-3.0.html" style="color: black;">here</a>.'
80         )
81         label_copyright.setAlignment(Qt.AlignCenter)
82         label_copyright.setTextFormat(Qt.RichText)
83         label_copyright.setOpenExternalLinks(True)
84
85         # === Arrange the widgets
86
87         self.setContentsMargins(10, 10, 10, 10)
88
89         vlay = QVBoxLayout()
90         vlay.setSpacing(20)
91         vlay.addWidget(label_title)
92         vlay.addWidget(label_version_info)
93         vlay.addWidget(label_info)
94         vlay.addWidget(label_copyright)
95
96         self.setLayout(vlay)

```

A.13 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
13  from PyQt5 import QtWidgets
14  from PyQt5.QtGui import QIntValidator, QKeyEvent, QKeySequence
15  from PyQt5.QtWidgets import QCheckBox, QGroupBox, QHBoxLayout, QShortcut, QSizePolicy, QSpacerItem, QVBoxLayout
16
17  from lintrans.gui.dialogs.misc import FixedSizeDialog
18  from lintrans.gui.settings import DisplaySettings
19
20
21  class SettingsDialog(FixedSizeDialog):
22      """An abstract superclass for other simple dialogs."""
23
24      def __init__(self, *args, **kwargs):
25          """Create the widgets and layout of the dialog, passing ``*args`` and ``**kwargs`` to super."""
26          super().__init__(*args, **kwargs)
27
28          # === Create the widgets
29
30          self.button_confirm = QtWidgets.QPushButton(self)
31          self.button_confirm.setText('Confirm')
32          self.button_confirm.clicked.connect(self.confirm_settings)
33          self.button_confirm.setToolTip('Confirm these new settings<br><b>(Ctrl + Enter)</b>')
34          QShortcut(QKeySequence('Ctrl+Return'), self).activated.connect(self.button_confirm.click)
35
36          self.button_cancel = QtWidgets.QPushButton(self)
37          self.button_cancel.setText('Cancel')
38          self.button_cancel.clicked.connect(self.reject)
39          self.button_cancel.setToolTip('Revert these settings<br><b>(Escape)</b>')
40
41          # === Arrange the widgets
42
43          self.setContentsMargins(10, 10, 10, 10)
44
45          self.hlay_buttons = QHBoxLayout()
46          self.hlay_buttons.setSpacing(20)
47          self.hlay_buttons.addItem(QSpacerItem(50, 5, hPolicy=QSizePolicy.Expanding, vPolicy=QSizePolicy.Minimum))
48          self.hlay_buttons.addWidget(self.button_cancel)
49          self.hlay_buttons.addWidget(self.button_confirm)
50
51          self.vlay_options = QVBoxLayout()
52          self.vlay_options.setSpacing(20)
53
54          self.vlay_all = QVBoxLayout()
55          self.vlay_all.setSpacing(20)
56          self.vlay_all.addLayout(self.vlay_options)
57          self.vlay_all.addLayout(self.hlay_buttons)
58
59          self.setLayout(self.vlay_all)
60
61      @abc.abstractmethod
62      def load_settings(self) -> None:
63          """Load the current settings into the widgets."""
64
65      @abc.abstractmethod
66      def confirm_settings(self) -> None:
67          """Confirm the settings chosen in the dialog."""
68
69
70  class DisplaySettingsDialog(SettingsDialog):

```



```

71     """The dialog to allow the user to edit the display settings."""
72
73     def __init__(self, *args, display_settings: DisplaySettings, **kwargs):
74         """Create the widgets and layout of the dialog.
75
76         :param DisplaySettings display_settings: The :class:`lintrans.gui.settings.DisplaySettings` object to mutate
77         """
78         super().__init__(*args, **kwargs)
79
80         self.display_settings = display_settings
81         self.setWindowTitle('Change display settings')
82
83         self.dict_checkboxes: dict[str, QCheckBox] = dict()
84
85         # == Create the widgets
86
87         # Basic stuff
88
89         self.checkbox_draw_background_grid = QCheckBox(self)
90         self.checkbox_draw_background_grid.setText('Draw &background grid')
91         self.checkbox_draw_background_grid.setToolTip(
92             'Draw the background grid (axes are always drawn)'
93         )
94         self.dict_checkboxes['b'] = self.checkbox_draw_background_grid
95
96         self.checkbox_draw_transformed_grid = QCheckBox(self)
97         self.checkbox_draw_transformed_grid.setText('Draw t&ransformed grid')
98         self.checkbox_draw_transformed_grid.setToolTip(
99             'Draw the transformed grid (vectors are handled separately)'
100        )
101        self.dict_checkboxes['r'] = self.checkbox_draw_transformed_grid
102
103        self.checkbox_draw_basis_vectors = QCheckBox(self)
104        self.checkbox_draw_basis_vectors.setText('Draw basis &vectors')
105        self.checkbox_draw_basis_vectors.setToolTip(
106            'Draw the transformed basis vectors'
107        )
108        self.dict_checkboxes['v'] = self.checkbox_draw_basis_vectors
109
110        # Animations
111
112        self.checkbox_smooththen_determinant = QCheckBox(self)
113        self.checkbox_smooththen_determinant.setText('&Smoothen determinant')
114        self.checkbox_smooththen_determinant.setToolTip(
115            'Smoothly animate the determinant transition during animation (if possible)'
116        )
117        self.dict_checkboxes['s'] = self.checkbox_smooththen_determinant
118
119        self.checkbox_applicative_animation = QCheckBox(self)
120        self.checkbox_applicative_animation.setText('&Applicative animation')
121        self.checkbox_applicative_animation.setToolTip(
122            'Animate the new transformation applied to the current one,\n'
123            'rather than just that transformation on its own'
124        )
125        self.dict_checkboxes['a'] = self.checkbox_applicative_animation
126
127        self.label_animation_pause_length = QtWidgets.QLabel(self)
128        self.label_animation_pause_length.setText('Animation pause length (ms)')
129        self.label_animation_pause_length.setToolTip(
130            'How many milliseconds to pause for in comma-separated animations'
131        )
132
133        self.lineedit_animation_pause_length = QtWidgets.QLineEdit(self)
134        self.lineedit_animation_pause_length.setValidator(QIntValidator(1, 999, self))
135
136        # Matrix info
137
138        self.checkbox_draw_determinant_parallelogram = QCheckBox(self)
139        self.checkbox_draw_determinant_parallelogram.setText('Draw &determinant parallelogram')
140        self.checkbox_draw_determinant_parallelogram.setToolTip(
141            'Shade the parallelogram representing the determinant of the matrix'
142        )
143        self.checkbox_draw_determinant_parallelogram.clicked.connect(self.update_gui)

```

```

144     self.dict_checkboxes['d'] = self.checkbox_draw_determinant_parallelogram
145
146     self.checkbox_show_determinant_value = QCheckBox(self)
147     self.checkbox_show_determinant_value.setText('Show de&eterminant value')
148     self.checkbox_show_determinant_value.setToolTip(
149         'Show the value of the determinant inside the parallelogram'
150     )
151     self.dict_checkboxes['t'] = self.checkbox_show_determinant_value
152
153     self.checkbox_draw_eigenvectors = QCheckBox(self)
154     self.checkbox_draw_eigenvectors.setText('Draw &eigenvectors')
155     self.checkbox_draw_eigenvectors.setToolTip('Draw the eigenvectors of the transformations')
156     self.dict_checkboxes['e'] = self.checkbox_draw_eigenvectors
157
158     self.checkbox_draw_eigenlines = QCheckBox(self)
159     self.checkbox_draw_eigenlines.setText('Draw eigen&lines')
160     self.checkbox_draw_eigenlines.setToolTip('Draw the eigenlines (invariant lines) of the transformations')
161     self.dict_checkboxes['l'] = self.checkbox_draw_eigenlines
162
163     # === Arrange the widgets in QGroupBoxes
164
165     # Basic stuff
166
167     self.vlay_groupbox_basic_stuff = QVBoxLayout()
168     self.vlay_groupbox_basic_stuff.setSpacing(20)
169     self.vlay_groupbox_basic_stuff.addWidget(self.checkbox_draw_background_grid)
170     self.vlay_groupbox_basic_stuff.addWidget(self.checkbox_draw_transformed_grid)
171     self.vlay_groupbox_basic_stuff.addWidget(self.checkbox_draw_basis_vectors)
172
173     self.groupbox_basic_stuff = QGroupBox('Basic stuff', self)
174     self.groupbox_basic_stuff.setLayout(self.vlay_groupbox_basic_stuff)
175
176     # Animations
177
178     self.hlay_animation_pause_length = QHBoxLayout()
179     self.hlay_animation_pause_length.addWidget(self.label_animation_pause_length)
180     self.hlay_animation_pause_length.addWidget(self.lineedit_animation_pause_length)
181
182     self.vlay_groupbox_animations = QVBoxLayout()
183     self.vlay_groupbox_animations.setSpacing(20)
184     self.vlay_groupbox_animations.addWidget(self.checkbox_smoother_determinant)
185     self.vlay_groupbox_animations.addWidget(self.checkbox_applicative_animation)
186     self.vlay_groupbox_animations.addLayout(self.hlay_animation_pause_length)
187
188     self.groupbox_animations = QGroupBox('Animations', self)
189     self.groupbox_animations.setLayout(self.vlay_groupbox_animations)
190
191     # Matrix info
192
193     self.vlay_groupbox_matrix_info = QVBoxLayout()
194     self.vlay_groupbox_matrix_info.setSpacing(20)
195     self.vlay_groupbox_matrix_info.addWidget(self.checkbox_draw_determinant_parallelogram)
196     self.vlay_groupbox_matrix_info.addWidget(self.checkbox_show_determinant_value)
197     self.vlay_groupbox_matrix_info.addWidget(self.checkbox_draw_eigenvectors)
198     self.vlay_groupbox_matrix_info.addWidget(self.checkbox_draw_eigenlines)
199
200     self.groupbox_matrix_info = QGroupBox('Matrix info', self)
201     self.groupbox_matrix_info.setLayout(self.vlay_groupbox_matrix_info)
202
203     # Now arrange the groupboxes
204     self.vlay_options.addWidget(self.groupbox_basic_stuff)
205     self.vlay_options.addWidget(self.groupbox_animations)
206     self.vlay_options.addWidget(self.groupbox_matrix_info)
207
208     # Finally, we load the current settings and update the GUI
209     self.load_settings()
210     self.update_gui()
211
212 def load_settings(self) -> None:
213     """Load the current display settings into the widgets."""
214     # Basic stuff
215     self.checkbox_draw_background_grid.setChecked(self.display_settings.draw_background_grid)
216     self.checkbox_draw_transformed_grid.setChecked(self.display_settings.draw_transformed_grid)

```

```

217         self.checkbox_draw_basis_vectors.setChecked(self.display_settings.draw_basis_vectors)
218
219     # Animations
220     self.checkbox_smoothen_determinant.setChecked(self.display_settings.smoothen_determinant)
221     self.checkbox_applicative_animation.setChecked(self.display_settings.applicative_animation)
222     self.lineEdit_animation_pause_length.setText(str(self.display_settings.animation_pause_length))
223
224     # Matrix info
225     self.checkbox_draw_determinant_parallelogram.setChecked(
226         ↪ self.display_settings.draw_determinant_parallelogram)
227     self.checkbox_show_determinant_value.setChecked(self.display_settings.show_determinant_value)
228     self.checkbox_draw_eigenvectors.setChecked(self.display_settings.draw_eigenvectors)
229     self.checkbox_draw_eigenlines.setChecked(self.display_settings.draw_eigenlines)
230
231     def confirm_settings(self) -> None:
232         """Build a :class:`lintrans.gui.settings.DisplaySettings` object and assign it."""
233         # Basic stuff
234         self.display_settings.draw_background_grid = self.checkbox_draw_background_grid.isChecked()
235         self.display_settings.draw_transformed_grid = self.checkbox_draw_transformed_grid.isChecked()
236         self.display_settings.draw_basis_vectors = self.checkbox_draw_basis_vectors.isChecked()
237
238         # Animations
239         self.display_settings.smoothen_determinant = self.checkbox_smoothen_determinant.isChecked()
240         self.display_settings.applicative_animation = self.checkbox_applicative_animation.isChecked()
241         self.display_settings.animation_pause_length = int(self.lineEdit_animation_pause_length.text())
242
243         # Matrix info
244         self.display_settings.draw_determinant_parallelogram =
245         ↪ self.checkbox_draw_determinant_parallelogram.isChecked()
246         self.display_settings.show_determinant_value = self.checkbox_show_determinant_value.isChecked()
247         self.display_settings.draw_eigenvectors = self.checkbox_draw_eigenvectors.isChecked()
248         self.display_settings.draw_eigenlines = self.checkbox_draw_eigenlines.isChecked()
249
250         self.accept()
251
252     def update_gui(self) -> None:
253         """Update the GUI according to other widgets in the GUI.
254         For example, this method updates which checkboxes are enabled based on the values of other checkboxes.
255         """
256         self.checkbox_show_determinant_value.setEnabled(self.checkbox_draw_determinant_parallelogram.isChecked())
257
258     def keyPressEvent(self, event: QKeyEvent) -> None:
259         """Handle a :class:`QKeyEvent` by manually activating toggling checkboxes.
260         Qt handles these shortcuts automatically and allows the user to do ``Alt + Key``
261         to activate a simple shortcut defined with ``&``. However, I like to be able to
262         just hit ``Key`` and have the shortcut activate.
263         """
264         letter = event.text().lower()
265         key = event.key()
266
267         if letter in self.dict_checkboxes:
268             self.dict_checkboxes[letter].animateClick()
269
270         # Return or keypad enter
271         elif key == 0x01000004 or key == 0x01000005:
272             self.button_confirm.click()
273
274         # Escape
275         elif key == 0x01000000:
276             self.button_cancel.click()
277
278         else:
279             event.ignore()

```

A.14 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 DefineAsAnExpressionDialog, DefineDialog, DefineNumericallyDialog,
13 ↵ DefineVisuallyDialog
14 from .misc import AboutDialog
15 from .settings import DisplaySettingsDialog
16
17 __all__ = ['DefineAsAnExpressionDialog', 'DefineDialog', 'DefineNumericallyDialog', 'DefineVisuallyDialog',
18           'AboutDialog', 'DisplaySettingsDialog']

```

A.15 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 from PyQt5.QtCore import Qt
12 from PyQt5.QtGui import QMouseEvent, QPainter, QPaintEvent
13
14 from .classes import VectorGridPlot
15 from lintrans.typing_ import MatrixType
16 from lintrans.gui.settings import DisplaySettings
17
18 class VisualizeTransformationWidget(VectorGridPlot):
19     """This class is the widget that is used in the main window to visualize transformations.
20
21     It handles all the rendering itself, and the only method that the user needs to
22     worry about is :meth:`visualize_matrix_transformation`, which allows you to visualize
23     the given matrix transformation.
24     """
25
26     def __init__(self, *args, display_settings: DisplaySettings, **kwargs):
27         """Create the widget and assign its display settings, passing ``*args`` and ``**kwargs`` to super."""
28         super().__init__(*args, **kwargs)
29
30         self.display_settings = display_settings
31
32     def visualize_matrix_transformation(self, matrix: MatrixType) -> None:
33         """Transform the grid by the given matrix.
34
35         .. warning:: This method does not call ``update()``. This must be done by the caller.
36
37         .. note::
38             This method transforms the background grid, not the basis vectors. This
39             means that it cannot be used to compose transformations. Compositions
40             should be done by the user.
41
42         :param MatrixType matrix: The matrix to transform by
43         """
44         self.point_i = (matrix[0][0], matrix[1][0])
45         self.point_j = (matrix[0][1], matrix[1][1])
46
47     def paintEvent(self, event: QPaintEvent) -> None:
48         """Handle a :class:`QPaintEvent` by drawing the background grid and the transformed grid.
49
50         The transformed grid is defined by the basis vectors i and j, which can
51

```

```

52         be controlled with the :meth:`visualize_matrix_transformation` method.
53         """
54         painter = QPainter()
55         painter.begin(self)
56
57         painter.setRenderHint(QPainter.Antialiasing)
58         painter.setBrush(Qt.NoBrush)
59
60         self.draw_background(painter, self.display_settings.draw_background_grid)
61
62         if self.display_settings.draw_eigenlines:
63             self.draw_eigenlines(painter)
64
65         if self.display_settings.draw_eigenvectors:
66             self.draw_eigenvectors(painter)
67
68         if self.display_settings.draw_determinant_parallelogram:
69             self.draw_determinant_parallelogram(painter)
70
71         if self.display_settings.show_determinant_value:
72             self.draw_determinant_text(painter)
73
74         if self.display_settings.draw_transformed_grid:
75             self.draw_transformed_grid(painter)
76
77         if self.display_settings.draw_basis_vectors:
78             self.draw_basis_vectors(painter)
79
80         painter.end()
81         event.accept()
82
83
84     class DefineVisuallyWidget(VisualizeTransformationWidget):
85         """This class is the widget that allows the user to visually define a matrix.
86
87         This is just the widget itself. If you want the dialog, use
88         :class:`lintrans.gui.dialogs.define_new_matrix.DefineVisuallyDialog`.
89         """
90
91         def __init__(self, *args, display_settings: DisplaySettings, **kwargs):
92             """Create the widget and enable mouse tracking. ``*args`` and ``**kwargs`` are passed to ``super()``."""
93             super().__init__(*args, display_settings=display_settings, **kwargs)
94
95             self.dragged_point: tuple[float, float] | None = None
96
97             # This is the distance that the cursor needs to be from the point to drag it
98             self.epsilon: int = 5
99
100        def mousePressEvent(self, event: QMouseEvent) -> None:
101            """Handle a QMouseEvent when the user pressed a button."""
102            mx = event.x()
103            my = event.y()
104            button = event.button()
105
106            if button != Qt.LeftButton:
107                event.ignore()
108                return
109
110            for point in (self.point_i, self.point_j):
111                px, py = self.canvas_coords(*point)
112                if abs(px - mx) <= self.epsilon and abs(py - my) <= self.epsilon:
113                    self.dragged_point = point[0], point[1]
114
115            event.accept()
116
117        def mouseReleaseEvent(self, event: QMouseEvent) -> None:
118            """Handle a QMouseEvent when the user release a button."""
119            if event.button() == Qt.LeftButton:
120                self.dragged_point = None
121                event.accept()
122            else:
123                event.ignore()
124

```

```

125     def mouseMoveEvent(self, event: QMouseEvent) -> None:
126         """Handle the mouse moving on the canvas."""
127         mx = event.x()
128         my = event.y()
129
130         if self.dragged_point is not None:
131             x, y = self.grid_coords(mx, my)
132
133             if self.dragged_point == self.point_i:
134                 self.point_i = x, y
135
136             elif self.dragged_point == self.point_j:
137                 self.point_j = x, y
138
139             self.dragged_point = x, y
140
141             self.update()
142
143             event.accept()
144
145             event.ignore()

```

A.16 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 . import classes
10 from .widgets import DefineVisuallyWidget, VisualizeTransformationWidget
11
12 __all__ = ['classes', 'DefineVisuallyWidget', 'VisualizeTransformationWidget']

```

A.17 gui/plots/classes.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 superclasses for plotting transformations."""
8
9  from __future__ import annotations
10
11 from abc import abstractmethod
12 from typing import Iterable
13
14 import numpy as np
15 from nptyping import Float, NDArray
16 from PyQt5.QtCore import QPoint, QRectF, Qt
17 from PyQt5.QtGui import QBrush, QColor, QPainter, QPainterPath, QPaintEvent, QPen, QWheelEvent
18 from PyQt5.QtWidgets import QWidget
19
20 from lintrans.typing_ import MatrixType
21
22
23 class BackgroundPlot(QWidget):
24     """This class provides a background for plotting, as well as setup for a Qt widget.
25
26     This class provides a background (untransformed) plane, and all the backend
27     details for a Qt application, but does not provide useful functionality. To
28     be useful, this class must be subclassed and behaviour must be implemented
29     by the subclass.

```

```

30
31 .. warning:: This class should never be directly instantiated, only subclassed.
32
33 .. note::
34     I would make this class have ``metaclass=abc.ABCMeta``, but I can't because it subclasses :class:`QWidget`,
35     and every superclass of a class must have the same metaclass, and :class:`QWidget` is not an abstract class.
36     """
37
38 default_grid_spacing: int = 85
39 minimum_grid_spacing: int = 5
40
41 def __init__(self, *args, **kwargs):
42     """Create the widget and setup backend stuff for rendering.
43
44     .. note:: ``*args`` and ``**kwargs`` are passed the superclass constructor (:class:`QWidget`).
45     """
46     super().__init__(*args, **kwargs)
47
48     self.setAutoFillBackground(True)
49
50     # Set the background to white
51     palette = self.palette()
52     palette.setColor(self.backgroundRole(), Qt.white)
53     self.setPalette(palette)
54
55     # Set the grid colour to grey and the axes colour to black
56     self.colour_background_grid = QColor('#808080')
57     self.colour_background_axes = QColor('#000000')
58
59     self.grid_spacing = BackgroundPlot.default_grid_spacing
60     self.width_background_grid: float = 0.3
61
62 @property
63 def canvas_origin(self) -> tuple[int, int]:
64     """Return the canvas coords of the grid origin.
65
66     The return value is intended to be unpacked and passed to a :meth:`QPainter.drawLine:iiii` call.
67
68     See :meth:`canvas_coords`.
69
70     :returns: The canvas coordinates of the grid origin
71     :rtype: tuple[int, int]
72     """
73     return self.width() // 2, self.height() // 2
74
75 def canvas_x(self, x: float) -> int:
76     """Convert an x coordinate from grid coords to canvas coords."""
77     return int(self.canvas_origin[0] + x * self.grid_spacing)
78
79 def canvas_y(self, y: float) -> int:
80     """Convert a y coordinate from grid coords to canvas coords."""
81     return int(self.canvas_origin[1] - y * self.grid_spacing)
82
83 def canvas_coords(self, x: float, y: float) -> tuple[int, int]:
84     """Convert a coordinate from grid coords to canvas coords.
85
86     This method is intended to be used like
87
88     .. code::
89
90         painter.drawLine(*self.canvas_coords(x1, y1), *self.canvas_coords(x2, y2))
91
92     or like
93
94     .. code::
95
96         painter.drawLine(*self.canvas_origin, *self.canvas_coords(x, y))
97
98     See :attr:`canvas_origin`.
99
100     :param float x: The x component of the grid coordinate
101     :param float y: The y component of the grid coordinate
102     :returns: The resultant canvas coordinates

```

```

103         :rtype: tuple[int, int]
104         """
105         return self.canvas_x(x), self.canvas_y(y)
106
107     def grid_corner(self) -> tuple[float, float]:
108         """Return the grid coords of the top right corner."""
109         return self.width() / (2 * self.grid_spacing), self.height() / (2 * self.grid_spacing)
110
111     def grid_coords(self, x: int, y: int) -> tuple[float, float]:
112         """Convert a coordinate from canvas coords to grid coords.
113
114         :param int x: The x component of the canvas coordinate
115         :param int y: The y component of the canvas coordinate
116         :returns: The resultant grid coordinates
117         :rtype: tuple[float, float]
118         """
119         # We get the maximum grid coords and convert them into canvas coords
120         return (x - self.canvas_origin[0]) / self.grid_spacing, (-y + self.canvas_origin[1]) / self.grid_spacing
121
122     @abstractmethod
123     def paintEvent(self, event: QPaintEvent) -> None:
124         """Handle a :class:`QPaintEvent`.
125
126         .. note:: This method is abstract and must be overridden by all subclasses.
127         """
128
129     def draw_background(self, painter: QPainter, draw_grid: bool) -> None:
130         """Draw the background grid.
131
132         .. note:: This method is just a utility method for subclasses to use to render the background grid.
133
134         :param QPainter painter: The painter to draw the background with
135         :param bool draw_grid: Whether to draw the grid lines
136         """
137         if draw_grid:
138             painter.setPen(QPen(self.colour_background_grid, self.width_background_grid))
139
140             # Draw equally spaced vertical lines, starting in the middle and going out
141             # We loop up to half of the width. This is because we draw a line on each side in each iteration
142             for x in range(self.width() // 2 + self.grid_spacing, self.width(), self.grid_spacing):
143                 painter.drawLine(x, 0, x, self.height())
144                 painter.drawLine(self.width() - x, 0, self.width() - x, self.height())
145
146             # Same with the horizontal lines
147             for y in range(self.height() // 2 + self.grid_spacing, self.height(), self.grid_spacing):
148                 painter.drawLine(0, y, self.width(), y)
149                 painter.drawLine(0, self.height() - y, self.width(), self.height() - y)
150
151             # Now draw the axes
152             painter.setPen(QPen(self.colour_background_axes, self.width_background_grid))
153             painter.drawLine(self.width() // 2, 0, self.width() // 2, self.height())
154             painter.drawLine(0, self.height() // 2, self.width(), self.height() // 2)
155
156     def wheelEvent(self, event: QWheelEvent) -> None:
157         """Handle a :class:`QWheelEvent` by zooming in or out of the grid."""
158         # angleDelta() returns a number of units equal to 8 times the number of degrees rotated
159         degrees = event.angleDelta() / 8
160
161         if degrees is not None:
162             new_spacing = max(1, self.grid_spacing + degrees.y())
163
164             if new_spacing >= self.minimum_grid_spacing:
165                 self.grid_spacing = new_spacing
166
167         event.accept()
168         self.update()
169
170
171 class VectorGridPlot(BackgroundPlot):
172     """This class represents a background plot, with vectors and their grid drawn on top.
173
174     This class should be subclassed to be used for visualization and matrix definition widgets.
175     All useful behaviour should be implemented by any subclass.

```



```

176
177 .. warning:: This class should never be directly instantiated, only subclassed.
178 """
179
180 def __init__(self, *args, **kwargs):
181     """Create the widget with ``point_i`` and ``point_j`` attributes.
182
183     .. note:: ``*args`` and ``**kwargs`` are passed to the superclass constructor (:class:`BackgroundPlot`).
184     """
185     super().__init__(*args, **kwargs)
186
187     self.point_i: tuple[float, float] = (1., 0.)
188     self.point_j: tuple[float, float] = (0., 1.)
189
190     self.colour_i = QColor('#0808d8')
191     self.colour_j = QColor('#e90000')
192     self.colour_eigen = QColor('#13cf00')
193     self.colour_text = QColor('#000000')
194
195     self.width_vector_line = 1.8
196     self.width_transformed_grid = 0.8
197
198     self.arrowhead_length = 0.15
199
200     self.max_parallel_lines = 150
201
202 @property
203 def matrix(self) -> MatrixType:
204     """Return the assembled matrix of the basis vectors."""
205     return np.array([
206         [self.point_i[0], self.point_j[0]],
207         [self.point_i[1], self.point_j[1]]
208     ])
209
210 @property
211 def det(self) -> float:
212     """Return the determinant of the assembled matrix."""
213     return float(np.linalg.det(self.matrix))
214
215 @property
216 def eigs(self) -> Iterable[tuple[float, NDArray[(1, 2), Float]]]:
217     """Return the eigenvalues and eigenvectors zipped together to be iterated over.
218
219     :rtype: Iterable[tuple[float, NDArray[(1, 2), Float]]]
220     """
221     values, vectors = np.linalg.eig(self.matrix)
222     return zip(values, vectors.T)
223
224 @abstractmethod
225 def paintEvent(self, event: QPaintEvent) -> None:
226     """Handle a :class:`QPaintEvent`.
227
228     .. note:: This method is abstract and must be overridden by all subclasses.
229     """
230
231 def draw_parallel_lines(self, painter: QPainter, vector: tuple[float, float], point: tuple[float, float]) ->
↳ None:
232     """Draw a set of evenly spaced grid lines parallel to ``vector`` intersecting ``point``.
233
234     :param QPainter painter: The painter to draw the lines with
235     :param vector: The vector to draw the grid lines parallel to
236     :type vector: tuple[float, float]
237     :param point: The point for the lines to intersect with
238     :type point: tuple[float, float]
239     """
240     max_x, max_y = self.grid_corner()
241     vector_x, vector_y = vector
242     point_x, point_y = point
243
244     # If the determinant is 0
245     if abs(vector_x * point_y - vector_y * point_x) < 1e-12:
246         rank = np.linalg.matrix_rank(
247             np.array([

```

```

248         [vector_x, point_x],
249         [vector_y, point_y]
250     ])
251 )
252
253 # If the matrix is rank 1, then we can draw the column space line
254 if rank == 1:
255     if abs(vector_x) < 1e-12:
256         painter.drawLine(self.width() // 2, 0, self.width() // 2, self.height())
257     elif abs(vector_y) < 1e-12:
258         painter.drawLine(0, self.height() // 2, self.width(), self.height() // 2)
259     else:
260         self.draw_oblique_line(painter, vector_y / vector_x, 0)
261
262 # If the rank is 0, then we don't draw any lines
263 else:
264     return
265
266 elif abs(vector_x) < 1e-12 and abs(vector_y) < 1e-12:
267     # If both components of the vector are practically 0, then we can't render any grid lines
268     return
269
270 # Draw vertical lines
271 elif abs(vector_x) < 1e-12:
272     painter.drawLine(self.canvas_x(0), 0, self.canvas_x(0), self.height())
273
274     for i in range(max(abs(int(max_x / point_x)), self.max_parallel_lines)):
275         painter.drawLine(
276             self.canvas_x((i + 1) * point_x),
277             0,
278             self.canvas_x((i + 1) * point_x),
279             self.height()
280         )
281         painter.drawLine(
282             self.canvas_x(-1 * (i + 1) * point_x),
283             0,
284             self.canvas_x(-1 * (i + 1) * point_x),
285             self.height()
286         )
287
288 # Draw horizontal lines
289 elif abs(vector_y) < 1e-12:
290     painter.drawLine(0, self.canvas_y(0), self.width(), self.canvas_y(0))
291
292     for i in range(max(abs(int(max_y / point_y)), self.max_parallel_lines)):
293         painter.drawLine(
294             0,
295             self.canvas_y((i + 1) * point_y),
296             self.width(),
297             self.canvas_y((i + 1) * point_y)
298         )
299         painter.drawLine(
300             0,
301             self.canvas_y(-1 * (i + 1) * point_y),
302             self.width(),
303             self.canvas_y(-1 * (i + 1) * point_y)
304         )
305
306 # If the line is oblique, then we can use y = mx + c
307 else:
308     m = vector_y / vector_x
309     c = point_y - m * point_x
310
311     self.draw_oblique_line(painter, m, 0)
312
313     # We don't want to overshoot the max number of parallel lines,
314     # but we should also stop looping as soon as we can't draw any more lines
315     for i in range(1, self.max_parallel_lines + 1):
316         if not self.draw_pair_of_oblique_lines(painter, m, i * c):
317             break
318
319 def draw_pair_of_oblique_lines(self, painter: QPainter, m: float, c: float) -> bool:
320     """Draw a pair of oblique lines, using the equation y = mx + c.

```

```

321
322     This method just calls :meth:`draw_oblique_line` with ``c`` and ``-c``,
323     and returns True if either call returned True.
324
325     :param QPainter painter: The painter to draw the vectors and grid lines with
326     :param float m: The gradient of the lines to draw
327     :param float c: The y-intercept of the lines to draw. We use the positive and negative versions
328     :returns bool: Whether we were able to draw any lines on the canvas
329     """
330     return any([
331         self.draw_oblique_line(painter, m, c),
332         self.draw_oblique_line(painter, m, -c)
333     ])
334
335 def draw_oblique_line(self, painter: QPainter, m: float, c: float) -> bool:
336     """Draw an oblique line, using the equation  $y = mx + c$ .
337
338     We only draw the part of the line that fits within the canvas, returning True if
339     we were able to draw a line within the boundaries, and False if we couldn't draw a line
340
341     :param QPainter painter: The painter to draw the vectors and grid lines with
342     :param float m: The gradient of the line to draw
343     :param float c: The y-intercept of the line to draw
344     :returns bool: Whether we were able to draw a line on the canvas
345     """
346     max_x, max_y = self.grid_corner()
347
348     # These variable names are shortened for convenience
349     # myi is max_y_intersection, mmyi is minus_max_y_intersection, etc.
350     myi = (max_y - c) / m
351     mmyi = (-max_y - c) / m
352     mxi = max_x * m + c
353     mmxi = -max_x * m + c
354
355     # The inner list here is a list of coords, or None
356     # If an intersection fits within the bounds, then we keep its coord,
357     # else it is None, and then gets discarded from the points list
358     # By the end, points is a list of two coords, or an empty list
359     points: list[tuple[float, float]] = [
360         x for x in [
361             (myi, max_y) if -max_x < myi < max_x else None,
362             (mmyi, -max_y) if -max_x < mmyi < max_x else None,
363             (max_x, mxi) if -max_y < mxi < max_y else None,
364             (-max_x, mmxi) if -max_y < mmxi < max_y else None
365         ] if x is not None
366     ]
367
368     # If no intersections fit on the canvas
369     if len(points) < 2:
370         return False
371
372     # If we can, then draw the line
373     else:
374         painter.drawLine(
375             *self.canvas_coords(*points[0]),
376             *self.canvas_coords(*points[1])
377         )
378         return True
379
380 def draw_transformed_grid(self, painter: QPainter) -> None:
381     """Draw the transformed version of the grid, given by the basis vectors.
382
383     .. note:: This method draws the grid, but not the basis vectors. Use :meth:`draw_basis_vectors` to draw
384     ↪ them.
385
386     :param QPainter painter: The painter to draw the grid lines with
387     """
388     # Draw all the parallel lines
389     painter.setPen(QPen(self.colour_i, self.width_transformed_grid))
390     self.draw_parallel_lines(painter, self.point_i, self.point_j)
391     painter.setPen(QPen(self.colour_j, self.width_transformed_grid))
392     self.draw_parallel_lines(painter, self.point_j, self.point_i)

```

```

393 def draw_arrowhead_away_from_origin(self, painter: QPainter, point: tuple[float, float]) -> None:
394     """Draw an arrowhead at ``point``, pointing away from the origin.
395
396     :param QPainter painter: The painter to draw the arrowhead with
397     :param point: The point to draw the arrowhead at, given in grid coords
398     :type point: tuple[float, float]
399     """
400     # This algorithm was adapted from a C# algorithm found at
401     # http://csharpshelper.com/blog/2014/12/draw-lines-with-arrowheads-in-c/
402
403     # Get the x and y coords of the point, and then normalize them
404     # We have to normalize them, or else the size of the arrowhead will
405     # scale with the distance of the point from the origin
406     x, y = point
407     vector_length = np.sqrt(x * x + y * y)
408
409     if vector_length < 1e-12:
410         return
411
412     nx = x / vector_length
413     ny = y / vector_length
414
415     # We choose a length and find the steps in the x and y directions
416     length = min(
417         self.arrowhead_length * self.default_grid_spacing / self.grid_spacing,
418         vector_length
419     )
420     dx = length * (-nx - ny)
421     dy = length * (nx - ny)
422
423     # Then we just plot those lines
424     painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x + dx, y + dy))
425     painter.drawLine(*self.canvas_coords(x, y), *self.canvas_coords(x - dy, y + dx))
426
427 def draw_position_vector(self, painter: QPainter, point: tuple[float, float], colour: QColor) -> None:
428     """Draw a vector from the origin to the given point.
429
430     :param QPainter painter: The painter to draw the position vector with
431     :param point: The tip of the position vector in grid coords
432     :type point: tuple[float, float]
433     :param QColor colour: The colour to draw the position vector in
434     """
435     painter.setPen(QPen(colour, self.width_vector_line))
436     painter.drawLine(*self.canvas_origin, *self.canvas_coords(*point))
437     self.draw_arrowhead_away_from_origin(painter, point)
438
439 def draw_basis_vectors(self, painter: QPainter) -> None:
440     """Draw arrowheads at the tips of the basis vectors.
441
442     :param QPainter painter: The painter to draw the basis vectors with
443     """
444     self.draw_position_vector(painter, self.point_i, self.colour_i)
445     self.draw_position_vector(painter, self.point_j, self.colour_j)
446
447 def draw_determinant_parallelogram(self, painter: QPainter) -> None:
448     """Draw the parallelogram of the determinant of the matrix.
449
450     :param QPainter painter: The painter to draw the parallelogram with
451     """
452     if self.det == 0:
453         return
454
455     path = QPainterPath()
456     path.moveTo(*self.canvas_origin)
457     path.lineTo(*self.canvas_coords(*self.point_i))
458     path.lineTo(*self.canvas_coords(self.point_i[0] + self.point_j[0], self.point_i[1] + self.point_j[1]))
459     path.lineTo(*self.canvas_coords(*self.point_j))
460
461     color = (16, 235, 253) if self.det > 0 else (253, 34, 16)
462     brush = QBrush(QColor(*color, alpha=128), Qt.SolidPattern)
463
464     painter.fillPath(path, brush)
465

```

```

466 def draw_determinant_text(self, painter: QPainter) -> None:
467     """Write the string value of the determinant in the middle of the parallelogram.
468
469     :param QPainter painter: The painter to draw the determinant text with
470     """
471     painter.setPen(QPen(self.colour_text, self.width_vector_line))
472
473     # We're building a QRect that encloses the determinant parallelogram
474     # Then we can center the text in this QRect
475     coords: list[tuple[float, float]] = [
476         (0, 0),
477         self.point_i,
478         self.point_j,
479         (
480             self.point_i[0] + self.point_j[0],
481             self.point_i[1] + self.point_j[1]
482         )
483     ]
484
485     xs = [t[0] for t in coords]
486     ys = [t[1] for t in coords]
487
488     top_left = QPoint(*self.canvas_coords(min(xs), max(ys)))
489     bottom_right = QPoint(*self.canvas_coords(max(xs), min(ys)))
490
491     rect = QRectF(top_left, bottom_right)
492
493     painter.drawText(
494         rect,
495         Qt.AlignHCenter | Qt.AlignVCenter,
496         f'{self.det:.2f}'
497     )
498
499 def draw_eigenvectors(self, painter: QPainter) -> None:
500     """Draw the eigenvectors of the displayed matrix transformation.
501
502     :param QPainter painter: The painter to draw the eigenvectors with
503     """
504     for value, vector in self.eigs:
505         x = value * vector[0]
506         y = value * vector[1]
507
508         if x.imag != 0 or y.imag != 0:
509             continue
510
511         self.draw_position_vector(painter, (x, y), self.colour_eigen)
512
513         # Now we need to draw the eigenvalue at the tip of the eigenvector
514
515         offset = 3
516         top_left: QPoint
517         bottom_right: QPoint
518         alignment_flags: int
519
520         if x >= 0 and y >= 0: # Q1
521             top_left = QPoint(self.canvas_x(x) + offset, 0)
522             bottom_right = QPoint(self.width(), self.canvas_y(y) - offset)
523             alignment_flags = Qt.AlignLeft | Qt.AlignBottom
524
525         elif x < 0 and y >= 0: # Q2
526             top_left = QPoint(0, 0)
527             bottom_right = QPoint(self.canvas_x(x) - offset, self.canvas_y(y) - offset)
528             alignment_flags = Qt.AlignRight | Qt.AlignBottom
529
530         elif x < 0 and y < 0: # Q3
531             top_left = QPoint(0, self.canvas_y(y) + offset)
532             bottom_right = QPoint(self.canvas_x(x) - offset, self.height())
533             alignment_flags = Qt.AlignRight | Qt.AlignTop
534
535         else: # Q4
536             top_left = QPoint(self.canvas_x(x) + offset, self.canvas_y(y) + offset)
537             bottom_right = QPoint(self.width(), self.height())
538             alignment_flags = Qt.AlignLeft | Qt.AlignTop

```

```
539         painter.setPen(QPen(self.colour_text, self.width_vector_line))
540         painter.drawText(QRectF(top_left, bottom_right), alignment_flags, f'{value:.2f}')
541
542
543     def draw_eigenlines(self, painter: QPainter) -> None:
544         """Draw the eigenlines (invariant lines).
545
546         :param QPainter painter: The painter to draw the eigenlines with
547         """
548         painter.setPen(QPen(self.colour_eigen, self.width_transformed_grid))
549
550         for value, vector in self.eigs:
551             if value.imag != 0:
552                 continue
553
554             x, y = vector
555
556             if x == 0:
557                 x_mid = int(self.width() / 2)
558                 painter.drawLine(x_mid, 0, x_mid, self.height())
559
560             elif y == 0:
561                 y_mid = int(self.height() / 2)
562                 painter.drawLine(0, y_mid, self.width(), y_mid)
563
564             else:
565                 self.draw_oblique_line(painter, y / x, 0)
```

B Testing code

B.1 matrices/test_rotation_matrices.py

```

1  # lintrans - The linear transformation visualizer
2  # Copyright (C) 2021-2022 D. Dyson (DoctorDalek1963)
3
4  # This program is licensed under GNU GPLv3, available here:
5  # <https://www.gnu.org/licenses/gpl-3.0.html>
6
7  """Test functions for rotation matrices."""
8
9  import numpy as np
10 import pytest
11
12 from lintrans.matrices import create_rotation_matrix
13 from lintrans.typing_ import MatrixType
14
15 angles_and_matrices: list[tuple[float, float, MatrixType]] = [
16     (0, 0, np.array([[1, 0], [0, 1]])),
17     (90, np.pi / 2, np.array([[0, -1], [1, 0]])),
18     (180, np.pi, np.array([[ -1, 0], [0, -1]])),
19     (270, 3 * np.pi / 2, np.array([[0, 1], [-1, 0]])),
20     (360, 2 * np.pi, np.array([[1, 0], [0, 1]])),
21
22     (45, np.pi / 4, np.array([
23         [np.sqrt(2) / 2, -1 * np.sqrt(2) / 2],
24         [np.sqrt(2) / 2, np.sqrt(2) / 2]
25     ])),
26     (135, 3 * np.pi / 4, np.array([
27         [-1 * np.sqrt(2) / 2, -1 * np.sqrt(2) / 2],
28         [np.sqrt(2) / 2, -1 * np.sqrt(2) / 2]
29     ])),
30     (225, 5 * np.pi / 4, np.array([
31         [-1 * np.sqrt(2) / 2, np.sqrt(2) / 2],
32         [-1 * np.sqrt(2) / 2, -1 * np.sqrt(2) / 2]
33     ])),
34     (315, 7 * np.pi / 4, np.array([
35         [np.sqrt(2) / 2, np.sqrt(2) / 2],
36         [-1 * np.sqrt(2) / 2, np.sqrt(2) / 2]
37     ])),
38
39     (30, np.pi / 6, np.array([
40         [np.sqrt(3) / 2, -1 / 2],
41         [1 / 2, np.sqrt(3) / 2]
42     ])),
43     (60, np.pi / 3, np.array([
44         [1 / 2, -1 * np.sqrt(3) / 2],
45         [np.sqrt(3) / 2, 1 / 2]
46     ])),
47     (120, 2 * np.pi / 3, np.array([
48         [-1 / 2, -1 * np.sqrt(3) / 2],
49         [np.sqrt(3) / 2, -1 / 2]
50     ])),
51     (150, 5 * np.pi / 6, np.array([
52         [-1 * np.sqrt(3) / 2, -1 / 2],
53         [1 / 2, -1 * np.sqrt(3) / 2]
54     ])),
55     (210, 7 * np.pi / 6, np.array([
56         [-1 * np.sqrt(3) / 2, 1 / 2],
57         [-1 / 2, -1 * np.sqrt(3) / 2]
58     ])),
59     (240, 4 * np.pi / 3, np.array([
60         [-1 / 2, np.sqrt(3) / 2],
61         [-1 * np.sqrt(3) / 2, -1 / 2]
62     ])),
63     (300, 10 * np.pi / 6, np.array([
64         [1 / 2, np.sqrt(3) / 2],
65         [-1 * np.sqrt(3) / 2, 1 / 2]
66     ])),
67     (330, 11 * np.pi / 6, np.array([

```

```

68         [np.sqrt(3) / 2, 1 / 2],
69         [-1 / 2, np.sqrt(3) / 2]
70     )))
71 ]
72
73
74 def test_create_rotation_matrix() -> None:
75     """Test that create_rotation_matrix() works with given angles and expected matrices."""
76     for degrees, radians, matrix in angles_and_matrices:
77         assert create_rotation_matrix(degrees, degrees=True) == pytest.approx(matrix)
78         assert create_rotation_matrix(radians, degrees=False) == pytest.approx(matrix)
79
80     assert create_rotation_matrix(-1 * degrees, degrees=True) == pytest.approx(np.linalg.inv(matrix))
81     assert create_rotation_matrix(-1 * radians, degrees=False) == pytest.approx(np.linalg.inv(matrix))

```

B.2 matrices/test_parse_and_validate_expression.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 :mod:`matrices.parse` module validation and parsing."""
8
9  import pytest
10
11  from lintrans.matrices.parse import (
12      MatrixParseError, find_sub_expressions, parse_matrix_expression, validate_matrix_expression
13  )
14  from lintrans.typing_ import MatrixParseList
15
16  expected_sub_expressions: list[tuple[str, list[str]]] = [
17      ('2(AB)^-1', ['AB']),
18      ('-3(A+B)^2-C(B^TA)^-1', ['A+B', 'B^TA']),
19      ('rot(45)', []),
20      ('()', []),
21      ('()', ['()']),
22      ('2.3A^-1(AB)^-1+(BC)^2', ['AB', 'BC']),
23      ('(2.3A^-1(AB)^-1+(BC)^2)', ['2.3A^-1(AB)^-1+(BC)^2']),
24  ]
25
26
27 def test_find_sub_expressions() -> None:
28     """Test the :func:`lintrans.matrices.parse.find_sub_expressions` function."""
29     for inp, output in expected_sub_expressions:
30         assert find_sub_expressions(inp) == output
31
32
33  valid_inputs: list[str] = [
34      'A', 'AB', '3A', '1.2A', '-3.4A', 'A^2', 'A^-1', 'A^{-1}',
35      'A^12', 'A^T', 'A^{5}', 'A^{T}', '4.3A^7', '9.2A^{18}', '0.1A'
36
37      'rot(45)', 'rot(12.5)', '3rot(90)',
38      'rot(135)^3', 'rot(51)^T', 'rot(-34)^-1',
39
40      'A+B', 'A+2B', '4.3A+9B', 'A^2+B^T', '3A^7+0.8B^{16}',
41      'A-B', '3A-4B', '3.2A^3-16.79B^T', '4.752A^{17}-3.32B^{36}',
42      'A-1B', '-A', '-1A'
43
44      '3A4B', 'A^TB', 'A^{T}B', '4A^6B^3',
45      '2A^{3}4B^5', '4rot(90)^3', 'rot(45)rot(13)',
46      'Arot(90)', 'AB^2', 'A^2B^2', '8.36A^T3.4B^12',
47
48      '3.5A^{4}5.6rot(19.2)^T-B^{-1}4.1C^5',
49
50      '(A)', '(AB)^-1', '2.3(3B^TA)^2', '-3.4(9D^{2}3F^{-1})^T+C', '(AB)(C)',
51      '3(rot(34)^-7A)^-1+B', '3A^2B+4A(B+C)^-1D^T-A(C(D+E)B)'
52  ]
53

```



```

54 invalid_inputs: list[str] = [
55     '', 'rot()', 'A^', 'A^1.2', 'A^{3.4}', '1,2A', 'ro(12)', '5', '12^2', '^T', '^12}', '.1A',
56     'A^{13}', 'A^3}', 'A^A', '^2', 'A--B', '--A', '+A', '--1A', 'A--B', 'A--1B', '.A', '1.A',
57     '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)',
58     '3)^2', '4(your mum)^T', 'rot()', 'rot(10.1.1)', 'rot(--2)',
59
60     'This is 100% a valid matrix expression, I swear'
61 ]
62
63
64 @pytest.mark.parametrize('inputs,output', [(valid_inputs, True), (invalid_inputs, False)])
65 def test_validate_matrix_expression(inputs: list[str], output: bool) -> None:
66     """Test the validate_matrix_expression() function."""
67     for inp in inputs:
68         assert validate_matrix_expression(inp) == output
69
70
71 expressions_and_parsed_expressions: list[tuple[str, MatrixParseList]] = [
72     # Simple expressions
73     ('A', [[(' ', 'A', ' ')]]),
74     ('A^2', [[(' ', 'A', '2')]]),
75     ('A^{2}', [[(' ', 'A', '2')]]),
76     ('3A', [[('3', 'A', ' ')]]),
77     ('1.4A^3', [[('1.4', 'A', '3')]]),
78     ('0.1A', [[('0.1', 'A', ' ')]]),
79     ('0.1A', [[('0.1', 'A', ' ')]]),
80     ('A^12', [[(' ', 'A', '12')]]),
81     ('A^234', [[(' ', 'A', '234')]]),
82
83     # Multiplications
84     ('A 0.1B', [[(' ', 'A', ' '), ('0.1', 'B', ' ')]]),
85     ('A^2 3B', [[(' ', 'A', '23'), (' ', 'B', ' ')]]),
86     ('4A^{3} 6B^2', [[('4', 'A', '3'), ('6', 'B', '2')]]),
87     ('4.2A^{T} 6.1B^{-1}', [[('4.2', 'A', 'T'), ('6.1', 'B', '-1')]]),
88     ('-1.2A^2 rot(45)^2', [[('1.2', 'A', '2'), (' ', 'rot(45)', '2')]]),
89     ('3.2A^T 4.5B^{5} 9.6rot(121.3)', [[('3.2', 'A', 'T'), ('4.5', 'B', '5'), ('9.6', 'rot(121.3)', ' ')]]),
90     ('-1.18A^{-2} 0.1B^{2} 9rot(-34.6)^{-1}', [[('1.18', 'A', '-2'), ('0.1', 'B', '2'), ('9', 'rot(-34.6)', '-1')]]),
91
92     # Additions
93     ('A + B', [[(' ', 'A', ' '), (' ', 'B', ' ')]]),
94     ('A + B - C', [[(' ', 'A', ' '), (' ', 'B', ' '), ('-1', 'C', ' ')]]),
95     ('A^2 + 0.5B', [[(' ', 'A', '2'), ('0.5', 'B', ' ')]]),
96     ('2A^3 + 8B^T - 3C^{-1}', [[('2', 'A', '3'), ('8', 'B', 'T'), ('-3', 'C', '-1')]]),
97     ('4.9A^2 - 3rot(134.2)^{-1} + 7.6B^8', [[('4.9', 'A', '2'), ('-3', 'rot(134.2)', '-1'), ('7.6', 'B', '8')]]),
98
99     # Additions with multiplication
100     ('2.14A^{3} 4.5rot(14.5)^{-1} + 8B^T - 3C^{-1}', [[('2.14', 'A', '3'), ('4.5', 'rot(14.5)', '-1'),
101     [ ('8', 'B', 'T'), ('-3', 'C', '-1') ] ]),
102     ('2.14A^{3} 4.5rot(14.5)^{-1} + 8.5B^T 5.97C^{14} - 3.14D^{-1} 6.7E^T',
103     [[('2.14', 'A', '3'), ('4.5', 'rot(14.5)', '-1'), ('8.5', 'B', 'T'), ('5.97', 'C', '14'),
104     [ ('-3.14', 'D', '-1'), ('6.7', 'E', 'T') ] ]),
105
106     # Parenthesized expressions
107     ('(AB)^{-1}', [[(' ', 'AB', '-1')]]),
108     ('-3(A+B)^2-C(B^TA)^{-1}', [[('3', 'A+B', '2'), ('-1', 'C', ' '), (' ', 'B^{T}A', '-1')]]),
109     ('2.3(3B^TA)^2', [[('2.3', '3B^{T}A', '2')]]),
110     ('-3.4(9D^{2}3F^{-1})^T+C', [[('3.4', '9D^{2}3F^{-1}', 'T'), (' ', 'C', ' ')]]),
111     ('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}',
112     [ '-1' ] ])]
113 ]
114
115 def test_parse_matrix_expression() -> None:
116     """Test the parse_matrix_expression() function."""
117     for expression, parsed_expression in expressions_and_parsed_expressions:
118         # Test it with and without whitespace
119         assert parse_matrix_expression(expression) == parsed_expression
120         assert parse_matrix_expression(expression.replace(' ', '')) == parsed_expression
121
122
123 def test_parse_error() -> None:
124     """Test that parse_matrix_expression() raises a MatrixParseError."""
125     for expression in invalid_inputs:

```

```

126         with pytest.raises(MatrixParseError):
127             parse_matrix_expression(expression)

```

B.3 matrices/matrix_wrapper/conftest.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  """A simple conftest.py containing some re-usable fixtures."""
8
9  import numpy as np
10 import pytest
11
12 from lintrans.matrices import MatrixWrapper
13
14
15 def get_test_wrapper() -> MatrixWrapper:
16     """Return a new MatrixWrapper object with some preset values."""
17     wrapper = MatrixWrapper()
18
19     root_two_over_two = np.sqrt(2) / 2
20
21     wrapper['A'] = np.array([[1, 2], [3, 4]])
22     wrapper['B'] = np.array([[6, 4], [12, 9]])
23     wrapper['C'] = np.array([[ -1, -3], [4, -12]])
24     wrapper['D'] = np.array([[13.2, 9.4], [-3.4, -1.8]])
25     wrapper['E'] = np.array([
26         [root_two_over_two, -1 * root_two_over_two],
27         [root_two_over_two, root_two_over_two]
28     ])
29     wrapper['F'] = np.array([[ -1, 0], [0, 1]])
30     wrapper['G'] = np.array([[np.pi, np.e], [1729, 743.631]])
31
32     return wrapper
33
34
35 @pytest.fixture
36 def test_wrapper() -> MatrixWrapper:
37     """Return a new MatrixWrapper object with some preset values."""
38     return get_test_wrapper()
39
40
41 @pytest.fixture
42 def new_wrapper() -> MatrixWrapper:
43     """Return a new MatrixWrapper with no initialized values."""
44     return MatrixWrapper()

```

B.4 matrices/matrix_wrapper/test_evaluate_expression.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 evaluate_expression() method."""
8
9  import numpy as np
10 from numpy import linalg as la
11 import pytest
12 from pytest import approx
13
14 from lintrans.matrices import MatrixWrapper, create_rotation_matrix
15 from lintrans.typing_ import MatrixType
16

```

```

17 from conftest import get_test_wrapper
18
19
20 def test_simple_matrix_addition(test_wrapper: MatrixWrapper) -> None:
21     """Test simple addition and subtraction of two matrices."""
22
23     # NOTE: We assert that all of these values are not None just to stop mypy complaining
24     # These values will never actually be None because they're set in the wrapper() fixture
25     # There's probably a better way do this, because this method is a bit of a bodge, but this works for now
26     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
27         test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
28         test_wrapper['G'] is not None
29
30     assert (test_wrapper.evaluate_expression('A+B') == test_wrapper['A'] + test_wrapper['B']).all()
31     assert (test_wrapper.evaluate_expression('E+F') == test_wrapper['E'] + test_wrapper['F']).all()
32     assert (test_wrapper.evaluate_expression('G+D') == test_wrapper['G'] + test_wrapper['D']).all()
33     assert (test_wrapper.evaluate_expression('C+C') == test_wrapper['C'] + test_wrapper['C']).all()
34     assert (test_wrapper.evaluate_expression('D+A') == test_wrapper['D'] + test_wrapper['A']).all()
35     assert (test_wrapper.evaluate_expression('B+C') == test_wrapper['B'] + test_wrapper['C']).all()
36
37     assert test_wrapper == get_test_wrapper()
38
39
40 def test_simple_two_matrix_multiplication(test_wrapper: MatrixWrapper) -> None:
41     """Test simple multiplication of two matrices."""
42     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
43         test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
44         test_wrapper['G'] is not None
45
46     assert (test_wrapper.evaluate_expression('AB') == test_wrapper['A'] @ test_wrapper['B']).all()
47     assert (test_wrapper.evaluate_expression('BA') == test_wrapper['B'] @ test_wrapper['A']).all()
48     assert (test_wrapper.evaluate_expression('AC') == test_wrapper['A'] @ test_wrapper['C']).all()
49     assert (test_wrapper.evaluate_expression('DA') == test_wrapper['D'] @ test_wrapper['A']).all()
50     assert (test_wrapper.evaluate_expression('ED') == test_wrapper['E'] @ test_wrapper['D']).all()
51     assert (test_wrapper.evaluate_expression('FD') == test_wrapper['F'] @ test_wrapper['D']).all()
52     assert (test_wrapper.evaluate_expression('GA') == test_wrapper['G'] @ test_wrapper['A']).all()
53     assert (test_wrapper.evaluate_expression('CF') == test_wrapper['C'] @ test_wrapper['F']).all()
54     assert (test_wrapper.evaluate_expression('AG') == test_wrapper['A'] @ test_wrapper['G']).all()
55
56     assert test_wrapper.evaluate_expression('A2B') == approx(test_wrapper['A'] @ (2 * test_wrapper['B']))
57     assert test_wrapper.evaluate_expression('2AB') == approx((2 * test_wrapper['A']) @ test_wrapper['B'])
58     assert test_wrapper.evaluate_expression('C3D') == approx(test_wrapper['C'] @ (3 * test_wrapper['D']))
59     assert test_wrapper.evaluate_expression('4.2E1.2A') == approx((4.2 * test_wrapper['E']) @ (1.2 *
60         ↪ test_wrapper['A']))
61
62     assert test_wrapper == get_test_wrapper()
63
64 def test_identity_multiplication(test_wrapper: MatrixWrapper) -> None:
65     """Test that multiplying by the identity doesn't change the value of a matrix."""
66     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
67         test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
68         test_wrapper['G'] is not None
69
70     assert (test_wrapper.evaluate_expression('I') == test_wrapper['I']).all()
71     assert (test_wrapper.evaluate_expression('AI') == test_wrapper['A']).all()
72     assert (test_wrapper.evaluate_expression('IA') == test_wrapper['A']).all()
73     assert (test_wrapper.evaluate_expression('GI') == test_wrapper['G']).all()
74     assert (test_wrapper.evaluate_expression('IG') == test_wrapper['G']).all()
75
76     assert (test_wrapper.evaluate_expression('EID') == test_wrapper['E'] @ test_wrapper['D']).all()
77     assert (test_wrapper.evaluate_expression('IED') == test_wrapper['E'] @ test_wrapper['D']).all()
78     assert (test_wrapper.evaluate_expression('EDI') == test_wrapper['E'] @ test_wrapper['D']).all()
79     assert (test_wrapper.evaluate_expression('IEIDI') == test_wrapper['E'] @ test_wrapper['D']).all()
80     assert (test_wrapper.evaluate_expression('EI^3D') == test_wrapper['E'] @ test_wrapper['D']).all()
81
82     assert test_wrapper == get_test_wrapper()
83
84
85 def test_simple_three_matrix_multiplication(test_wrapper: MatrixWrapper) -> None:
86     """Test simple multiplication of two matrices."""
87     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
88         test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \

```

```

89         test_wrapper['G'] is not None
90
91     assert (test_wrapper.evaluate_expression('ABC') == test_wrapper['A'] @ test_wrapper['B'] @
92     ↪ test_wrapper['C']).all()
93     assert (test_wrapper.evaluate_expression('ACB') == test_wrapper['A'] @ test_wrapper['C'] @
94     ↪ test_wrapper['B']).all()
95     assert (test_wrapper.evaluate_expression('BAC') == test_wrapper['B'] @ test_wrapper['A'] @
96     ↪ test_wrapper['C']).all()
97     assert (test_wrapper.evaluate_expression('EFG') == test_wrapper['E'] @ test_wrapper['F'] @
98     ↪ test_wrapper['G']).all()
99     assert (test_wrapper.evaluate_expression('DAC') == test_wrapper['D'] @ test_wrapper['A'] @
100    ↪ test_wrapper['C']).all()
101     assert (test_wrapper.evaluate_expression('GAE') == test_wrapper['G'] @ test_wrapper['A'] @
102    ↪ test_wrapper['E']).all()
103     assert (test_wrapper.evaluate_expression('FAG') == test_wrapper['F'] @ test_wrapper['A'] @
104    ↪ test_wrapper['G']).all()
105     assert (test_wrapper.evaluate_expression('GAF') == test_wrapper['G'] @ test_wrapper['A'] @
106    ↪ test_wrapper['F']).all()
107
108     assert test_wrapper == get_test_wrapper()
109
110 def test_matrix_inverses(test_wrapper: MatrixWrapper) -> None:
111     """Test the inverses of single matrices."""
112     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
113     test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
114     test_wrapper['G'] is not None
115
116     assert (test_wrapper.evaluate_expression('A^{-1}') == la.inv(test_wrapper['A'])).all()
117     assert (test_wrapper.evaluate_expression('B^{-1}') == la.inv(test_wrapper['B'])).all()
118     assert (test_wrapper.evaluate_expression('C^{-1}') == la.inv(test_wrapper['C'])).all()
119     assert (test_wrapper.evaluate_expression('D^{-1}') == la.inv(test_wrapper['D'])).all()
120     assert (test_wrapper.evaluate_expression('E^{-1}') == la.inv(test_wrapper['E'])).all()
121     assert (test_wrapper.evaluate_expression('F^{-1}') == la.inv(test_wrapper['F'])).all()
122     assert (test_wrapper.evaluate_expression('G^{-1}') == la.inv(test_wrapper['G'])).all()
123
124     assert (test_wrapper.evaluate_expression('A^{-1}') == la.inv(test_wrapper['A'])).all()
125     assert (test_wrapper.evaluate_expression('B^{-1}') == la.inv(test_wrapper['B'])).all()
126     assert (test_wrapper.evaluate_expression('C^{-1}') == la.inv(test_wrapper['C'])).all()
127     assert (test_wrapper.evaluate_expression('D^{-1}') == la.inv(test_wrapper['D'])).all()
128     assert (test_wrapper.evaluate_expression('E^{-1}') == la.inv(test_wrapper['E'])).all()
129     assert (test_wrapper.evaluate_expression('F^{-1}') == la.inv(test_wrapper['F'])).all()
130     assert (test_wrapper.evaluate_expression('G^{-1}') == la.inv(test_wrapper['G'])).all()
131
132     assert test_wrapper == get_test_wrapper()
133
134 def test_matrix_powers(test_wrapper: MatrixWrapper) -> None:
135     """Test that matrices can be raised to integer powers."""
136     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
137     test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
138     test_wrapper['G'] is not None
139
140     assert (test_wrapper.evaluate_expression('A^2') == la.matrix_power(test_wrapper['A'], 2)).all()
141     assert (test_wrapper.evaluate_expression('B^4') == la.matrix_power(test_wrapper['B'], 4)).all()
142     assert (test_wrapper.evaluate_expression('C^{12}') == la.matrix_power(test_wrapper['C'], 12)).all()
143     assert (test_wrapper.evaluate_expression('D^{12}') == la.matrix_power(test_wrapper['D'], 12)).all()
144     assert (test_wrapper.evaluate_expression('E^8') == la.matrix_power(test_wrapper['E'], 8)).all()
145     assert (test_wrapper.evaluate_expression('F^{-6}') == la.matrix_power(test_wrapper['F'], -6)).all()
146     assert (test_wrapper.evaluate_expression('G^{-2}') == la.matrix_power(test_wrapper['G'], -2)).all()
147
148     assert test_wrapper == get_test_wrapper()
149
150 def test_matrix_transpose(test_wrapper: MatrixWrapper) -> None:
151     """Test matrix transpositions."""
152     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
153     test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
154     test_wrapper['G'] is not None
155
156     assert (test_wrapper.evaluate_expression('A^T') == test_wrapper['A'].T).all()
157     assert (test_wrapper.evaluate_expression('B^T') == test_wrapper['B'].T).all()
158     assert (test_wrapper.evaluate_expression('C^T') == test_wrapper['C'].T).all()

```

```

154     assert (test_wrapper.evaluate_expression('D^{T}') == test_wrapper['D'].T).all()
155     assert (test_wrapper.evaluate_expression('E^{T}') == test_wrapper['E'].T).all()
156     assert (test_wrapper.evaluate_expression('F^{T}') == test_wrapper['F'].T).all()
157     assert (test_wrapper.evaluate_expression('G^{T}') == test_wrapper['G'].T).all()
158
159     assert (test_wrapper.evaluate_expression('A^T') == test_wrapper['A'].T).all()
160     assert (test_wrapper.evaluate_expression('B^T') == test_wrapper['B'].T).all()
161     assert (test_wrapper.evaluate_expression('C^T') == test_wrapper['C'].T).all()
162     assert (test_wrapper.evaluate_expression('D^T') == test_wrapper['D'].T).all()
163     assert (test_wrapper.evaluate_expression('E^T') == test_wrapper['E'].T).all()
164     assert (test_wrapper.evaluate_expression('F^T') == test_wrapper['F'].T).all()
165     assert (test_wrapper.evaluate_expression('G^T') == test_wrapper['G'].T).all()
166
167     assert test_wrapper == get_test_wrapper()
168
169
170 def test_rotation_matrices(test_wrapper: MatrixWrapper) -> None:
171     """Test that 'rot(angle)' can be used in an expression."""
172     assert (test_wrapper.evaluate_expression('rot(90)') == create_rotation_matrix(90)).all()
173     assert (test_wrapper.evaluate_expression('rot(180)') == create_rotation_matrix(180)).all()
174     assert (test_wrapper.evaluate_expression('rot(270)') == create_rotation_matrix(270)).all()
175     assert (test_wrapper.evaluate_expression('rot(360)') == create_rotation_matrix(360)).all()
176     assert (test_wrapper.evaluate_expression('rot(45)') == create_rotation_matrix(45)).all()
177     assert (test_wrapper.evaluate_expression('rot(30)') == create_rotation_matrix(30)).all()
178
179     assert (test_wrapper.evaluate_expression('rot(13.43)') == create_rotation_matrix(13.43)).all()
180     assert (test_wrapper.evaluate_expression('rot(49.4)') == create_rotation_matrix(49.4)).all()
181     assert (test_wrapper.evaluate_expression('rot(-123.456)') == create_rotation_matrix(-123.456)).all()
182     assert (test_wrapper.evaluate_expression('rot(963.245)') == create_rotation_matrix(963.245)).all()
183     assert (test_wrapper.evaluate_expression('rot(-235.24)') == create_rotation_matrix(-235.24)).all()
184
185     assert test_wrapper == get_test_wrapper()
186
187
188 def test_multiplication_and_addition(test_wrapper: MatrixWrapper) -> None:
189     """Test multiplication and addition of matrices together."""
190     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
191         test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
192         test_wrapper['G'] is not None
193
194     assert (test_wrapper.evaluate_expression('AB+C') ==
195             test_wrapper['A'] @ test_wrapper['B'] + test_wrapper['C']).all()
196     assert (test_wrapper.evaluate_expression('DE-D') ==
197             test_wrapper['D'] @ test_wrapper['E'] - test_wrapper['D']).all()
198     assert (test_wrapper.evaluate_expression('FD+AB') ==
199             test_wrapper['F'] @ test_wrapper['D'] + test_wrapper['A'] @ test_wrapper['B']).all()
200     assert (test_wrapper.evaluate_expression('BA-DE') ==
201             test_wrapper['B'] @ test_wrapper['A'] - test_wrapper['D'] @ test_wrapper['E']).all()
202
203     assert (test_wrapper.evaluate_expression('2AB+3C') ==
204             (2 * test_wrapper['A'] @ test_wrapper['B'] + (3 * test_wrapper['C'])).all()
205     assert (test_wrapper.evaluate_expression('4D7.9E-1.2A') ==
206             (4 * test_wrapper['D'] @ (7.9 * test_wrapper['E']) - (1.2 * test_wrapper['A'])).all()
207
208     assert test_wrapper == get_test_wrapper()
209
210
211 def test_complicated_expressions(test_wrapper: MatrixWrapper) -> None:
212     """Test evaluation of complicated expressions."""
213     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
214         test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
215         test_wrapper['G'] is not None
216
217     assert (test_wrapper.evaluate_expression('-3.2A^T 4B^{ -1} 6C^{ -1} + 8.1D^{2} 3.2E^4') ==
218             (-3.2 * test_wrapper['A'].T) @ (4 * la.inv(test_wrapper['B'])) @ (6 * la.inv(test_wrapper['C']))
219             + (8.1 * la.matrix_power(test_wrapper['D'], 2)) @ (3.2 * la.matrix_power(test_wrapper['E'], 4))).all()
220
221     assert (test_wrapper.evaluate_expression('53.6D^{2} 3B^T - 4.9F^{2} 2D + A^3 B^{-1}') ==
222             (53.6 * la.matrix_power(test_wrapper['D'], 2)) @ (3 * test_wrapper['B'].T)
223             - (4.9 * la.matrix_power(test_wrapper['F'], 2)) @ (2 * test_wrapper['D'])
224             + la.matrix_power(test_wrapper['A'], 3) @ la.inv(test_wrapper['B'])).all()
225
226     assert test_wrapper == get_test_wrapper()

```

```

227
228
229 def test_parenthesized_expressions(test_wrapper: MatrixWrapper) -> None:
230     """Test evaluation of parenthesized expressions."""
231     assert test_wrapper['A'] is not None and test_wrapper['B'] is not None and test_wrapper['C'] is not None and \
232         test_wrapper['D'] is not None and test_wrapper['E'] is not None and test_wrapper['F'] is not None and \
233         test_wrapper['G'] is not None
234
235     assert (test_wrapper.evaluate_expression('(A^T)^2') == la.matrix_power(test_wrapper['A'].T, 2)).all()
236     assert (test_wrapper.evaluate_expression('(B^T)^3') == la.matrix_power(test_wrapper['B'].T, 3)).all()
237     assert (test_wrapper.evaluate_expression('(C^T)^4') == la.matrix_power(test_wrapper['C'].T, 4)).all()
238     assert (test_wrapper.evaluate_expression('(D^T)^5') == la.matrix_power(test_wrapper['D'].T, 5)).all()
239     assert (test_wrapper.evaluate_expression('(E^T)^6') == la.matrix_power(test_wrapper['E'].T, 6)).all()
240     assert (test_wrapper.evaluate_expression('(F^T)^7') == la.matrix_power(test_wrapper['F'].T, 7)).all()
241     assert (test_wrapper.evaluate_expression('(G^T)^8') == la.matrix_power(test_wrapper['G'].T, 8)).all()
242
243     assert (test_wrapper.evaluate_expression('(rot(45)^1)^T') == create_rotation_matrix(45).T).all()
244     assert (test_wrapper.evaluate_expression('(rot(45)^2)^T') == la.matrix_power(create_rotation_matrix(45),
245     ↪ 2).T).all()
246     assert (test_wrapper.evaluate_expression('(rot(45)^3)^T') == la.matrix_power(create_rotation_matrix(45),
247     ↪ 3).T).all()
248     assert (test_wrapper.evaluate_expression('(rot(45)^4)^T') == la.matrix_power(create_rotation_matrix(45),
249     ↪ 4).T).all()
250     assert (test_wrapper.evaluate_expression('(rot(45)^5)^T') == la.matrix_power(create_rotation_matrix(45),
251     ↪ 5).T).all()
252
253     assert (test_wrapper.evaluate_expression('D^3(A+6.2F-0.397G^TE)^-2+A') ==
254         la.matrix_power(test_wrapper['D'], 3) @ la.matrix_power(
255             test_wrapper['A'] + 6.2 * test_wrapper['F'] - 0.397 * test_wrapper['G'].T @ test_wrapper['E'],
256             -2
257         ) + test_wrapper['A']).all()
258
259     assert (test_wrapper.evaluate_expression('-1.2F^{3}4.9D^T(A^2(B+3E^TF)^-1)^2') ==
260         -1.2 * la.matrix_power(test_wrapper['F'], 3) @ (4.9 * test_wrapper['D'].T) @
261         la.matrix_power(
262             la.matrix_power(test_wrapper['A'], 2) @ la.matrix_power(
263                 test_wrapper['B'] + 3 * test_wrapper['E'].T @ test_wrapper['F'],
264                 -1
265             ),
266             2
267         ).all()
268
269 def test_value_errors(test_wrapper: MatrixWrapper) -> None:
270     """Test that evaluate_expression() raises a ValueError for any malformed input."""
271     invalid_expressions = ['', '+', '-', 'This is not a valid expression', '3+4',
272         'A+2', 'A^', '^2', 'A^-', 'At', 'A^t', '3^2']
273
274     for expression in invalid_expressions:
275         with pytest.raises(ValueError):
276             test_wrapper.evaluate_expression(expression)
277
278 def test_linalgerror() -> None:
279     """Test that certain expressions raise np.linalg.LinAlgError."""
280     matrix_a: MatrixType = np.array([
281         [0, 0],
282         [0, 0]
283     ])
284
285     matrix_b: MatrixType = np.array([
286         [1, 2],
287         [1, 2]
288     ])
289
290     wrapper = MatrixWrapper()
291     wrapper['A'] = matrix_a
292     wrapper['B'] = matrix_b
293
294     assert (wrapper.evaluate_expression('A') == matrix_a).all()
295     assert (wrapper.evaluate_expression('B') == matrix_b).all()
296
297     with pytest.raises(np.linalg.LinAlgError):

```

```

296         wrapper.evaluate_expression('A^-1')
297
298     with pytest.raises(np.linalg.LinAlgError):
299         wrapper.evaluate_expression('B^-1')
300
301     assert (wrapper['A'] == matrix_a).all()
302     assert (wrapper['B'] == matrix_b).all()

```

B.5 matrices/matrix_wrapper/test_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  """Test the miscellaneous methods of the MatrixWrapper class."""
8
9  from lintrans.matrices import MatrixWrapper
10
11
12  def test_get_expression(test_wrapper: MatrixWrapper) -> None:
13      """Test the get_expression method of the MatrixWrapper class."""
14      test_wrapper['N'] = 'A^2'
15      test_wrapper['O'] = '4B'
16      test_wrapper['P'] = 'A+C'
17
18      test_wrapper['Q'] = 'N^-1'
19      test_wrapper['R'] = 'P-40'
20      test_wrapper['S'] = 'NOP'
21
22      assert test_wrapper.get_expression('A') is None
23      assert test_wrapper.get_expression('B') is None
24      assert test_wrapper.get_expression('C') is None
25      assert test_wrapper.get_expression('D') is None
26      assert test_wrapper.get_expression('E') is None
27      assert test_wrapper.get_expression('F') is None
28      assert test_wrapper.get_expression('G') is None
29
30      assert test_wrapper.get_expression('N') == 'A^2'
31      assert test_wrapper.get_expression('O') == '4B'
32      assert test_wrapper.get_expression('P') == 'A+C'
33
34      assert test_wrapper.get_expression('Q') == 'N^-1'
35      assert test_wrapper.get_expression('R') == 'P-40'
36      assert test_wrapper.get_expression('S') == 'NOP'

```

B.6 matrices/matrix_wrapper/test_setitem_and_getitem.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  import numpy as np
10 from numpy import linalg as la
11 import pytest
12 from typing import Any
13
14 from lintrans.matrices import MatrixWrapper
15 from lintrans.typing_ import MatrixType
16
17 valid_matrix_names = 'ABCDEFGHJKLMNPQRSTUVWXYZ'
18 invalid_matrix_names = ['bad name', '123456', 'Th15 Is an 1nV@l1D n@m3', 'abc', 'a']
19

```

```

20 test_matrix: MatrixType = np.array([[1, 2], [4, 3]])
21
22
23 def test_basic_get_matrix(new_wrapper: MatrixWrapper) -> None:
24     """Test MatrixWrapper().__getitem__()."""
25     for name in valid_matrix_names:
26         assert new_wrapper[name] is None
27
28     assert (new_wrapper['I'] == np.array([[1, 0], [0, 1]])).all()
29
30
31 def test_get_name_error(new_wrapper: MatrixWrapper) -> None:
32     """Test that MatrixWrapper().__getitem__() raises a NameError if called with an invalid name."""
33     for name in invalid_matrix_names:
34         with pytest.raises(NameError):
35             _ = new_wrapper[name]
36
37
38 def test_basic_set_matrix(new_wrapper: MatrixWrapper) -> None:
39     """Test MatrixWrapper().__setitem__()."""
40     for name in valid_matrix_names:
41         new_wrapper[name] = test_matrix
42         assert (new_wrapper[name] == test_matrix).all()
43
44         new_wrapper[name] = None
45         assert new_wrapper[name] is None
46
47
48 def test_set_expression(test_wrapper: MatrixWrapper) -> None:
49     """Test that MatrixWrapper.__setitem__() can accept a valid expression."""
50     test_wrapper['N'] = 'A^2'
51     test_wrapper['O'] = 'BA+2C'
52     test_wrapper['P'] = 'E^T'
53     test_wrapper['Q'] = 'C^-1B'
54     test_wrapper['R'] = 'A^{2}3B'
55     test_wrapper['S'] = 'N^-1'
56     test_wrapper['T'] = 'PQP^-1'
57
58     with pytest.raises(TypeError):
59         test_wrapper['U'] = 'A+1'
60
61     with pytest.raises(TypeError):
62         test_wrapper['V'] = 'K'
63
64     with pytest.raises(TypeError):
65         test_wrapper['W'] = 'L^2'
66
67     with pytest.raises(TypeError):
68         test_wrapper['X'] = 'M^-1'
69
70
71 def test_simple_dynamic_evaluation(test_wrapper: MatrixWrapper) -> None:
72     """Test that expression-defined matrices are evaluated dynamically."""
73     test_wrapper['N'] = 'A^2'
74     test_wrapper['O'] = '4B'
75     test_wrapper['P'] = 'A+C'
76
77     assert (test_wrapper['N'] == test_wrapper.evaluate_expression('A^2')).all()
78     assert (test_wrapper['O'] == test_wrapper.evaluate_expression('4B')).all()
79     assert (test_wrapper['P'] == test_wrapper.evaluate_expression('A+C')).all()
80
81     assert (test_wrapper.evaluate_expression('N^2 + 3O') ==
82             la.matrix_power(test_wrapper.evaluate_expression('A^2'), 2) +
83             3 * test_wrapper.evaluate_expression('4B')
84             ).all()
85     assert (test_wrapper.evaluate_expression('P^-1 - 3N0^2') ==
86             la.inv(test_wrapper.evaluate_expression('A+C')) -
87             (3 * test_wrapper.evaluate_expression('A^2')) @
88             la.matrix_power(test_wrapper.evaluate_expression('4B'), 2)
89             ).all()
90
91     test_wrapper['A'] = np.array([
92         [19, -21.5],

```



```

93         [84, 96.572]
94     ])
95     test_wrapper['B'] = np.array([
96         [-0.993, 2.52],
97         [1e10, 0]
98     ])
99     test_wrapper['C'] = np.array([
100         [0, 19512],
101         [1.414, 19]
102     ])
103
104     assert (test_wrapper['N'] == test_wrapper.evaluate_expression('A^2')).all()
105     assert (test_wrapper['O'] == test_wrapper.evaluate_expression('4B')).all()
106     assert (test_wrapper['P'] == test_wrapper.evaluate_expression('A+C')).all()
107
108     assert (test_wrapper.evaluate_expression('N^2 + 30') ==
109             la.matrix_power(test_wrapper.evaluate_expression('A^2'), 2) +
110             3 * test_wrapper.evaluate_expression('4B')
111             ).all()
112     assert (test_wrapper.evaluate_expression('P^-1 - 3N^2') ==
113             la.inv(test_wrapper.evaluate_expression('A+C')) -
114             (3 * test_wrapper.evaluate_expression('A^2')) @
115             la.matrix_power(test_wrapper.evaluate_expression('4B'), 2)
116             ).all()
117
118
119 def test_recursive_dynamic_evaluation(test_wrapper: MatrixWrapper) -> None:
120     """Test that dynamic evaluation works recursively."""
121     test_wrapper['N'] = 'A^2'
122     test_wrapper['O'] = '4B'
123     test_wrapper['P'] = 'A+C'
124
125     test_wrapper['Q'] = 'N^-1'
126     test_wrapper['R'] = 'P-4O'
127     test_wrapper['S'] = 'NOP'
128
129     assert test_wrapper['Q'] == pytest.approx(test_wrapper.evaluate_expression('A^-2'))
130     assert test_wrapper['R'] == pytest.approx(test_wrapper.evaluate_expression('A + C - 16B'))
131     assert test_wrapper['S'] == pytest.approx(test_wrapper.evaluate_expression('A^{2}4BA + A^{2}4BC'))
132
133
134 def test_set_identity_error(new_wrapper: MatrixWrapper) -> None:
135     """Test that MatrixWrapper().__setitem__() raises a NameError when trying to assign to the identity matrix."""
136     with pytest.raises(NameError):
137         new_wrapper['I'] = test_matrix
138
139
140 def test_set_name_error(new_wrapper: MatrixWrapper) -> None:
141     """Test that MatrixWrapper().__setitem__() raises a NameError when trying to assign to an invalid name."""
142     for name in invalid_matrix_names:
143         with pytest.raises(NameError):
144             new_wrapper[name] = test_matrix
145
146
147 def test_set_type_error(new_wrapper: MatrixWrapper) -> None:
148     """Test that MatrixWrapper().__setitem__() raises a TypeError when trying to set a non-matrix."""
149     invalid_values: list[Any] = [
150         12,
151         [1, 2, 3, 4, 5],
152         [[1, 2], [3, 4]],
153         True,
154         24.3222,
155         'This is totally a matrix, I swear',
156         MatrixWrapper,
157         MatrixWrapper(),
158         np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]]),
159         np.eye(100)
160     ]
161
162     for value in invalid_values:
163         with pytest.raises(TypeError):
164             new_wrapper['M'] = value

```

B.7 gui/test_dialog_utility_functions.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 utility functions for GUI dialog boxes."""
8
9  import numpy as np
10 import pytest
11
12 from lintrans.gui.dialogs.define_new_matrix import is_valid_float, round_float
13
14 valid_floats: list[str] = [
15     '0', '1', '3', '-2', '123', '-208', '1.2', '-3.5', '4.252634', '-42362.352325',
16     '1e4', '-2.59e3', '4.13e-6', '-5.5244e-12'
17 ]
18
19 invalid_floats: list[str] = [
20     '', 'pi', 'e', '1.2.3', '1,2', '-', '.', 'None', 'no', 'yes', 'float'
21 ]
22
23
24 @pytest.mark.parametrize('inputs,output', [(valid_floats, True), (invalid_floats, False)])
25 def test_is_valid_float(inputs: list[str], output: bool) -> None:
26     """Test the is_valid_float() function."""
27     for inp in inputs:
28         assert is_valid_float(inp) == output
29
30
31 def test_round_float() -> None:
32     """Test the round_float() function."""
33     expected_values: list[tuple[float, int, str]] = [
34         (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'),
35         (np.sqrt(2) / 2, 5, '0.70711'), (-1 * np.sqrt(2) / 2, 5, '-0.70711'),
36         (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'),
37         (1.23456789, 2, '1.23'), (1.23456789, 3, '1.235'), (1.23456789, 4, '1.2346'), (1.23456789, 5, '1.23457'),
38         (12345.678, 1, '12345.7'), (12345.678, 2, '12345.68'), (12345.678, 3, '12345.678'),
39     ]
40
41     for num, precision, answer in expected_values:
42         assert round_float(num, precision) == answer

```