I. REFERENCE DOCUMENT FOR COMPANION MATRIX IMPLEMENTATION - Ardana Labs internal circulation - Current draft does not have dimensionless polynomial in unknown x_k. **Update**, actually our dimensionlessness and numerical stability tricks all rely on polynomial denormalization which doesn't fly with companion matrix; document will be updated when I think of ways to recover some numerical stability.

I.1. Rootfinding by sticking the companion matrix into an eigenvalue solver

$\forall n \in \mathbb{N}$, forall p in the real polynomials of degree n arranged such that the leading coefficient is 1, the **companion matrix** of p called $C(p)$ is an $n \times n$ matrix. Its characteristic polynomial happens to be p. The eigenvalues of $C(p)$ are the roots of p. That is, since $\texttt{det}(C(p) - \lambda I) = p(\lambda)$, the solutions λ of $\texttt{det}(C(p) - \lambda I) = 0$ are an equal set to the solutions x of $p(x) = 0$.

I.1.1. We read invariant equation of Egorov 2019 as a polynomial, twice

For amplification coefficient A and balances $x := (x_1, ..., x_n)$

$$I_D := D \mapsto D^{n+1} + (A + n^{-n})n^{2n}(\Pi x_i)D + -An^{2n}(\Pi x_i)\Sigma x_i$$
$$\forall k \in 1..n, I_k := x_k \mapsto x_k^2 + \left(\Sigma_{i\neq k}x_i + (\frac{1}{An^n} - 1)D\right) x_k + \frac{-D^{n+1}}{An^{2n}\Pi_{i\neq k}x_i}$$

A more thorough treatment is in audit.pdf `III.2 rootfinding` with complete derivation from Egorov 2019 in `V.1. Appendix A`. The current document is just an implementation guide for the proof of concept/test.

The current document however will cover material that is lacking from the current draft of `audit.pdf`, being Nick's idea about **dimensionlessness**. The idea is, if you squint, a sum of balances of different assets all have the unit `asset`, as does D. We'll call this unit `\$`. A cursory glance at I_D shows that its units are `\$ ^ (n+1)` (The Π term is `\$ ^ n` and the Σ term is `\$`). **TODO**: dimensional analysis of I_k. The trick I *was* going to recommend was dividing by a factor that is also of unit `\$ ^ (n+1)` so that I_D becomes **dimensionless**. However, this would amount to **denormalizing** the polynomials, which the companion matrix definition cannot abide. (Polynomial normalization is what I performed in `V.1 Appendix A`, rewriting any $ax^2 + bx + c$ as $x^2 + (b/a)x + (c/a)$ under the equivalence relation defined by that polynomial's zeros). Since $k\texttt{det}(A - \lambda I) = \texttt{det}(kA - k\lambda I)$, there should be a way of recovering numerical tricks that may be required to keep the intermediary calculations from blowing out of `Double`. However, those will be saved for a later draft of this document.

I.1.2. The companion matrix for our polynomials

Let $p := x \mapsto p(x)$ be a polynomial of degree n defined by vector of coefficients $c := (c0, ..., c_{n-1})$ such $x \mapsto x^n + c_{n-1}x^{n-1} + ... + c_1x + c_0$, i.e. the coefficient 1 on the leading term **is ommitted** from the list c. Given like this, we can rewrite I_D as simply the list $(-An^{2n}(\Pi x_i)\Sigma x_i, (A + n^{-n})n^{2n}\Pi x_i, 0, ..., 0) = (S, T, 0, ..., 0)$ such that when there are n balances in the balance sheet there are $n - 2$ 0s at the end of the list. Similarly, we write I_k as simply the list $(\frac{-D^{n+1}}{An^{2n}\Pi_{i\neq k}x_i}, \Sigma_{i\neq k}x_i + (\frac{1}{An^n} - 1)D) = (U, V)$. The function $\forall n \in \mathbb{N}, C : \mathbb{R}^n \to \mathbb{M}_{n,n}$ (where polynomials are represented as vectors containing coefficients) is given on wikipedia, but here I will reproduce for $n \in \{2, 3, 4\}$.

First, however, I will do I_k, because it's quadratic.

If you recall that $I_K = (U, V)$, then

$$C(I_k) = \begin{bmatrix} 0 & -U \\ 1 & -V \end{bmatrix}$$

This will be the case for all sizes of balance sheets n, where once more $U := D, x_1, ..., x_{k-1}, x_{k+1}, ..., x_n \mapsto \frac{-D^{n+1}}{An^{2n}\Pi_{i \neq k} x_i}$ and $V := D, x_1, ..., x_{k-1}, x_{k+1}, ..., x_n \mapsto \Sigma_{i \neq k} x_i + (\frac{1}{An^n} - 1)D$.

Let I_D^n be the invariant polynomial I_D when there are n assets on the balance sheet (which is actually an $n+1$ -degree polynomial). Then,

$$\begin{aligned}
C(I_D^2) = C((S, T, 0)) &= \begin{bmatrix} 0 & 0 & -S \\ 1 & 0 & -T \\ 0 & 1 & 0 \end{bmatrix} \\
C(I_D^3) = C((S, T, 0, 0)) &= \begin{bmatrix} 0 & 0 & 0 & -S \\ 1 & 0 & 0 & -T \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \\
C(I_D^4) = C((S, T, 0, 0, 0)) &= \begin{bmatrix} 0 & 0 & 0 & 0 & -S \\ 1 & 0 & 0 & 0 & -T \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}
\end{aligned}$$

Where recall $S := x \mapsto -An^{2n}(\Pi x_i)\Sigma x_i$ and $T := x \mapsto (A + n^{-n})n^{2n}\Pi x_i$

```haskell
-- identity :: Nat -> [[Double]] (identity matrix)
companionID :: Int -> Double -> Double -> [[Double]]
companionID n s t = [
  (replicate n 0) ++ [- s],
  [1] ++ (replicate (n - 1) 0) ++ [- t]
  ]
  ++
  (tail $ take n $ identity (n + 1))

companionIxi :: Int -> Double -> Double [[Double]]
companionIxi n u v [
  [0, - v], [1, - u]
]
```

I.1.3. LAPACK

There exists a haskell wrapper around lapack

Morgan provided a breakdown of considerations comparing the lapack wrapper's eigen-solver with our own newton's method implementation (sometime in September, I think):

Here are some things to consider in considering using any algorithm implementation (first party or third party) to solve the invariant equation.

1. That algorithm will have limited precision and conditions for finding accurate solutions to our particular problems. Are those conditions met by the current invariant problem? Will they be met by future invariant problems?

LAPACK:

- Here is a page about the precision of its eigenvalue solver
- Margin of error is 3.1e-1

What about accuracy? Is the solver guaranteed to be accurate? What is the algorithm for the approximation? What are the assumptions of this algorithm? I don't know the answers to these questions.

Newton:

- Conditions for precision and accuracy are not exactly known.
- We could learn more in simulations

2. What platforms support that algorithm implementation?

- **LAPACK**: Haskell but not Plutus
- **Newton**: Haskell and Plutus

3. What platforms do we need the algorithm implementation to run on?

- Just Haskell, just on our own machines

4. Who supports the implementation and what kind of support is available to us?

LAPACK: actively maintained on GitHub with 72 contributors and 2,295 commits; issues on GitHub have about 50% rate of engagement; unknown what support is available

Haskell LAPACK: actively maintained by Dr. Henning Thielemann; unknown what level of support is available

Newton: it's all on us

5. How has the implementation been tested?

LAPACK: has half a million lines of test code

Haskell LAPACK: has 7.5k lines of unit tests

Newton: it's all on us and the testing is currently not sufficient

6. What is the performance like on expected problem sizes?

LAPACK: unknown

Newton: unknown, but sufficient, based on property tests

I.1.4. `Eigen`

`Eigen` is a C++ library that is widely trusted. There exists a wrapper for haskell.

I think I can figure out how to recover eigenvalues from `matrixQ` and `matrixR` here

I.1.5. Home rolled eigensolver

This book comes very highly recommended to me by someone I trust a lot. Page 347-426 is chapter 7 on unsymmetric eigenvalue problems.

Here's a pile of wikipedia pages

- Inverse iteration

- Jenkins-Traub
- QR algorithm

The case against

> If I were an attacker I'd want people doing as much custom code as possible, because I'm better at it than them, but I'm not better at it than the library authors. (a friend of mine)

Standard engineer wisdom is not to roll your own, because a distributed community can find bugs more adequately than your team.

The case for

General purpose algorithms are frequently less than 100% accurate on specific problems. Less than 100% is ok for a lot of scientific applications, but not for defi.