Gekko Roadmap 2025

Thomas Thomsen, September 8, 2025

Summary

Gekko is a timeseries-oriented programming language and environment, among other things used for handling timeseries data ("data wrangling") and for solving and analyzing economic models. Since Gekko started out around 2008, data science languages/platforms like for instance Python or R have become prevalent, and in the longer run, it would therefore make sense for Gekko to offer its features also in the form of a package inside such a "host" language. The point is that there is an overlap between what Gekko offers and such languages – for instance the handling of strings, lists, vectors/matrices, etc. – and those features could just as well (and better) be handled in the host data science language, letting Gekko focus on its "value-added" features, like for instance timeseries commands and functions, print "operators", model equation decomposition, etc.

Gekko in the form as a "Gekko-package" would be more viable in the longer run than Gekko solely as a stand-alone application ("Gekko-exe"), simply because data wrangling and -analysis (and modelling) has a tendency to move towards data science languages. Many users, particularly new or younger users, would prefer to use existing skills regarding for instance string/list/matrix manipulations in the data science language, only having to learn new syntax/logic regarding the value-added parts of Gekko. Another advantage of offering Gekko as a Gekko-package is that it would aid any possible migration of existing systems of Gekko-exe programs into the chosen host language, for instance Python.

One could envision the following spectrum of possibilities, from the rather simple to the most ambitious:

a1. Implement a simple Gekko-package in for instance Python, as a thin layer over Gekko-exe, continuing to use C# and the .NET Framework. This would be Windows-only.

a2. Integrate further into, say, Python, including syntax. But keep the Gekko source code in the C#.NET Framework. This would still be Windows-only.

a3. Migrate Gekko-exe to .NET Core (successor to .NET Framework). This would allow Gekko-exe to run smoothly on Windows pc's for a long time. It would also make it possible to run both Gekko-exe and Gekko-package on Linux or Mac, provided that Gekko popup windows such as plotting and decomposition etc. are also ported.

a4. Translate the value-added parts of the Gekko source code from C# to, say, C++ or Rust (for speed and platform independence).

Regarding (a1), Gekko commands could be sent off from inside Python in the form of simple text strings, still using Gekko syntax. This is how the existing Gekcel (Gekko add-in for Excel) component works, stating Gekko commands in VBA as text strings. From this thin layer, the syntax could gradually be integrated more and more deeply into, say, the Python syntax proper.

Recommendation

As mentioned, there is a whole spectrum of possibilities. But to keep the process relatively simple, and in order to progress in a stepwise fashion, the following is recommended, mirroring the (a1)-(a4) bullet list above. It is assumed here that Python is chosen as host language.

b1. Try out the simple Gekko-package (bullet point (a1) above), a thin Python layer over the existing Gekko-exe. This already works as a demo and presupposes very little work, and the user will for instance – from inside Python – be able to load a databank and a model (using Gekko commands as text strings), and decompose equations using the familiar decomposition windows. New users do not have to use a Gekko installer or stand-alone application, but can just "pip install" Gekko in Python and be up and running. This would only run on Windows computers, though, not Linux or Mac. More about this in section 2.1.

b2. Gradually experiment with deeper Python syntax integration (bullet point (a2) above), offering to write the most used Gekko commands directly in Python syntax. This would include deeper integration regarding loops, functions/procedures, conditional statements (if-then-else), and similar control structures. Possibly, the Gekko databanks could be implemented as dataframes, for instance in the Apache Arrow format. Some integration regarding array-series syntax/logic and GAMSPy (GAMS-package in Python) would also be expected. If at this point existing Gekko-exe users prefer to migrate their systems of command files to run as a Gekko-package inside Python, this might be a good opportunity to do this (automatic translators could be provided). More about this in section 2.2.

b3. While working on (b1) and (b2), Gekko-exe is kept up to date, functional, and released regularly, but it would probably be a worth-while investment to migrate the C# source code from the .NET Framework to .NET Core (bullet point (a3) above). This would make sure that both Gekko-exe and Gekko-package will run smoothly on Windows computers for many years to come, and the migration from the .NET Framework to .NET Core ought to be manageable resource-wise. In order to run Gekko-exe or Gekko-package on Linux or Mac, the Gekko popup windows like plotting and decomposition could migrate to for instance cross-platform Avalonia, or become browser-based like Plotly or similar. More about this in section 2.3.

b4. After these steps and experiences, more ambitious possibilities regarding Gekko-package could be considered (bullet point (a4) above), for instance translating the value-added parts of the C# code to C++ or Rust. More about this in section 2.4.

In addition to the above, at some point it is advisable to update the Gekko source code documentation, especially the value-added parts. This would make it easier to fix bugs and develop Gekko, by people other than the main Gekko developer. An existential threat to Gekko-exe would be if the program were not able to open or run reliably due to for instance a new Windows version (or security updates regarding an existing Windows version), combined with the Gekko editor and main developer having left the project. This risk is assessed in more detail in sections 3.9 and 3.10.

Contents

1 Preface

This paper was originally named “Gekko risk assessment”, originating from a concern of some users about the longer-term stability of the Gekko project and software. Since that, the contents of the paper have shifted from focusing mainly on risk mitigation to also addressing the future development of Gekko. If Gekko-exe, the current Gekko version, is not maintained and kept up to date regarding operating systems and other things, there is a risk that one day, the software refuses to open or fails to run stably (for instance following the installation of a newer Windows version). This is a risk for existing Gekko-exe users, of course, but there is another risk. Namely, that Gekko slowly fades into the obsolete/legacy/deprecated category, because Gekko as a project does not adapt to evolving trends. As reviewed in the following section 1.1, many users – especially new users – prefer software like Gekko to be offered in the form of a package inside a host (data science) language like for instance Python. An important element regarding the Gekko project is of course to retain users, including new users, and in the light of this, the paper was rewritten from being mostly about Gekko-exe risk assessments to also be about the future of Gekko more generally. More about dependence on individuals in section 3.9, and on existential threats regarding Gekko in section 3.10.

The paper consists of the two-page summary/cover, followed by this preface and the following two sections on recent software trends + a visual depiction of a possible roadmap for Gekko. Chapter 2 elaborates on the roadmap and the summary points (b1)-(b4), describing different levels of ambition and resource use. Chapter 3 provides even more details regarding the different possibilities.

1.1 Software trends, including data science software

First of all, security becomes more and more important, because of increasingly sophisticated cyberattack techniques, resulting in, for instance, data breaches. Therefore, running software on an updated and modern platform is important.

In addition, there is a general trend towards web-based and cloud-based, because web-based applications are easy to deploy in many kinds of environments (Windows/Linux/Mac/tablet/mobile, etc.), and cloud-based implies “always-on”, as long as there is an internet connection. Nowadays, browser programs run quite fast, too.[1] In the very long run, installing software with an installer tied to an operating system may become a thing of the past, with users instead expecting to be able to run software instantly as “web apps”, possibly in the cloud.

Regarding data science platforms in particular, these involve number crunching to some extent, and often quite large data sets. Data science can of course be done cloud-based, but quite often it is preferred to use the user’s pc for the number crunching, and sometimes the

[1] Technologies like the V8 browser engine for JavaScript and WebAssembly mean that a browser-side script does not necessarily run slow anymore.

users prefer to have the data “at hand” locally, for instance as local files, local servers, local databases, etc.[2]

There is no denying that open-source Python is the most popular data science platform, typically combined with packages like NumPy (arrays), Pandas/Polars[3] (dataframes) and others. What pushes Python ahead of the competition is also the fact that Python is by far the most used “host” language for machine learning. So the machine learning platforms typically piggyback the Python syntax, making it possible to define and run the machine learning models via Python syntax, and also use Python to access and filter data etc. But the inner parts of these machine learning platforms are not written in Python (which is too slow and does not have static types[4]), but rather in fast close-to-the-metal system languages like for instance C/C++ or Rust (the same applies to the source code of NumPy, Pandas, Polars, PyTorch, TensorFlow, etc.).

Python is a very clean language in itself. However, because it acts as a glue language for machine learning and many other things, the number of Python packages is vast, and the popularity of Python is not expected to diminish anytime soon. An alternative to Python would be open-source R, but R is becoming a bit more like a niche language for statistics (for which it is very entrenched) and data wrangling/visualization, and not so much a general purpose language. Else there is the newer open-source language Julia, which offers a Python-like syntax, but with speed comparable to C/C++, Rust, etc. The problem with Julia, though, is that it, too, is more like a niche language for now (competing with MATLAB and R), and still not making the inroads regarding Python that the developers probably had expected.

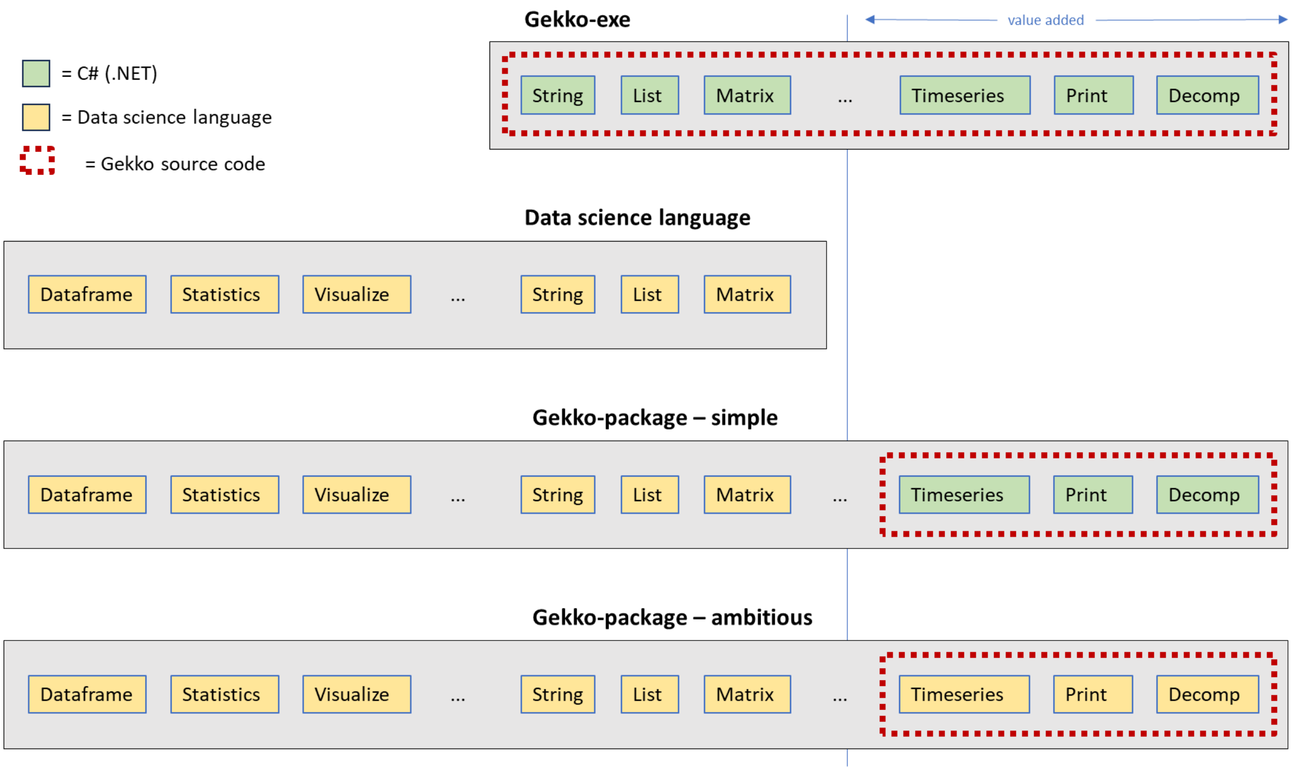
If software trends are not taken into account, a particular software package tends to stagnate and die out. Without really addressing the popularity of an ecosystem like Python, Gekko as a stand-alone application (Gekko-exe) would lose new users, existing users would start to migrate away, and sooner or later, Gekko would end up being considered a legacy application that runs legacy code/systems (it could of course also be possible, if Gekko-exe ends up being a legacy system, that users simply stay on Gekko-exe as long as possible, in order to avoid the costs of migrating to something else). The following section delves a bit more into the details of how to envision the Gekko-as-a-package philosophy.

[2] For government or other large institutions, hosting data in the cloud on for instance Amazon or Azure servers outside of the institution may be problematic because of data security, so for such institutions there are limits to the use of cloud, unless the cloud solutions are hosted “on-premise” (physical on-site server, not using an internet connection, and sometimes in the form of a “private cloud” solution).
[3] Polars was released around 2020 as a Pandas competitor written entirely in Rust, and with a focus on cleaner syntax and more speed.
[4] We shall return the question of static vs. dynamic types in section 3.2.

1.2 A possible roadmap for the future of Gekko

In the two cover pages, a recommendation regarding the future of Gekko was lined up. This section offers more details regarding this. We will begin with the following diagram:

Figure 1: Gekko-exe vs. Gekko-package



The existing Gekko-exe stand-alone application is shown in the first row at the top, containing modules that handle strings, lists, matrices, timeseries, prints, decompositions, etc. All of this is written in C#.NET Framework (green boxes: all of which are Gekko source code, as shown with the red dotted container).

The second row in the diagram shows the data science language (for instance Python), which also offers string, list and matrix handling, but in addition handles dataframes, statistics, visualizations and much more. Between Gekko-exe and the data science language, there are common modules like string handling, list handling and matrix handling, and if Gekko-exe is migrated to a Gekko-package, the host language could just take over these Gekko parts (and be more capable regarding them, too). As shown with the blue double arrow at the top, the value-added parts of Gekko, when envisioned inside a host language, are shown to the right: more specifically the timeseries, printing and decomposition parts (and more).

The third row in the diagram (corresponds to bullet points (b1) and (b2) in the summary) shows a simple or relatively simple Gekko-package, where the host language takes care of some of Gekko’s features (here: strings, lists, matrices), whereas Gekko offers other features (timeseries/print/decomp) in the form of a package. This Gekko-package would still be written in the C#.NET Framework (green boxes), and there are different options regarding how to do this, and whether .NET Core (bullet point (b3) in the summary) and Linux/Mac ought to be supported. Migrating to .NET Core would in itself benefit Gekko in terms of performance and security, especially after 2030, when the .NET Framework is expected to be considered legacy with no new features added.

The last row in the diagram (corresponds to bullet point (b4) in the summary) shows a development where the C# source code has been translated to another system language, for instance C++ or Rust, making the source code run faster and making cross-platform support easier.[5] If a host language like Julia was chosen, it could in principle be considered to translate the Gekko C# source code into Julia itself, but one problem with that is that Julia does not have static types (cf. section 3.2). If a faster-Python language/platform like the open-source Mojo takes off in a big way, translating the Gekko C# source code into the proposed static-type Python variant would be feasible.

As also mentioned in the summary, if existing Gekko users would like to port their existing Gekko programs/systems to for instance Python, such a transition could start out soon after the simple Gekko-package is released. Alternatively, users could wait for deeper Python integration, and migrate at that point. Migrating ought to be possible to do gradually, because running Gekko-exe in combination with Gekko-package would be possible, as long as Gekko-exe and Gekko-package can transfer data via Gekko databanks, dataframes or other data formats.

[5] It should be mentioned, though, that C# already runs quite fast, and often close to the speeds of for instance C++ or Rust.

2 Details regarding summary points b1-b4

2.1 Ad b1: Simple Gekko-package

A simple Gekko-package inside, say, Python as a host language is quite simple and straightforward to get up and running using another Python package called Python.NET. A downside to this would be that – unless migrating to .NET Core has already been accomplished (cf. section 2.3) – the simple Python package would be Windows-only. In many ways, a simple Gekko-package for Python would be similar to the existing Gekcel add-in for Excel, in which VBA is the "host language".

The following is an example of a simple string-based Python-Gekko program, showing first a plot window (second-to-last line) and then a decomposition window (last line):

```
import clr  # Python.NET (pythonnet)
clr.AddReference("...path...//Gekko.exe")
from Gekko import Python
gk = Python()
t1 = 2028
t2 = 2040
gk.Run("option folder working = '...path...';")
gk.Run("reset; ")
gk.Run(f"time {t1} {t2};")
gk.Run("read makro.gbk;")
gk.Run("model <gms> makro.zip;")
gk.Run("plot qBNP;")
gk.Run("decomp <d> qBNP from E_qBNP;")
```

The first four lines are for setting up Python.NET, and in the second line it is seen that it is just the normal `Gekko.exe` that is being referenced. But then in the following two lines, the time period is set, using standard Python syntax. These two dates, `t1` and `t2`, are subsequently used in the line `gk.Run(f"time {t1} {t2};")`, where `gk` stands for "Gekko" and it should be noted that – for readability – a so-called Python f-string is used, allowing easy string interpolation using {}-curlies. So, as it is seen, the time period `t1`-`t2` is easily transferred from Python proper to Gekko, and the same could be done regarding other settings, like databank and model pathnames, etc.[6]

2.2 Ad b2: More integrated Gekko-package

A deeper Python integration would be the possibility to write for instance `gk.Time(t1, t2)` instead of `gk.Run(f"time {t1} {t2};")`, where `Time()` is a specially constructed function (method) in the Gekko-package, and where `t1` and `t2` could possibly also be integers, strings or even specially defined Gekko-date-types with inbuilt frequency information. Such deeper

[6] Gekko uses similar string interpolation, like `time {%t1} {%t2};` where `%t1` and `%t2` are scalar variables. Distinguishing should be obvious enough due to Gekko's `%` sigil.

integration could be provided gradually step by step, starting with the most commonly used Gekko commands/functions.

One challenge when using a string-based thin interface is program control structure, like for instance loops. In Gekko one can write

```
%x = 0;
for val %i = 1 to 100;
    %x += %i;
end;
```

to sum up integers 1 to 100. But you would not be able to run this in Python as the following program:

```
gk.Run("%x = 0;")
gk.Run("for val %i = 1 to 100;")
gk.Run("    %x += %i;")
gk.Run("end;")
```

The second statement would fail, because Gekko expects an `end` to end the `for`. Therefore, in such a case, either the last three statements must be condensed like:

```
gk.Run("for val %i = 1 to 100; %x += %i; end;"),
```

or Python itself delivers the looping:

```
gk.Run("%x = 0;")
for i in range(1, 101):
    gk.Run(f"%x += {i};")
```

It should be emphasized that Gekko looping is quite often just looping over for instance lists of strings, so in practice performing the looping and other control structure would typically be relatively easy to transfer into Python proper.[7] Having Python handle the control structures – and therefore necessarily also the indentation level – is advised if possible, because it will make the Gekko-Python programs easier to understand.

Gekko is a so-called DSL (domain specific language), with its own syntax. Many software projects nowadays use a so-called embedded DSL (called eDSL) together with a flexible host language like for instance Python. When using an eDSL, normal Python syntax is used, but objects and methods are given special meaning, and operators are often ”hijacked” (via operator overloading). For instance, the following Python program:

[7] As mentioned before, such capabilities, like for instance list looping, are not considered a value-added part of Gekko, so performing them in Python is advised in any case. Conditional Gekko statements (if-then-else-end) are perhaps the most tricky part, but possibly a `gk.If()` function could be provided, testing if the string of Gekko code returns true or false, and then Python proper could handle that, using for instance `if gk.If("%t == 2010;")` to test if the Gekko scalar `%t` has the value 2010.

```
import gamspy as gp
m = gp.Container()
i = gp.Set(m, 'i', records=['i1','i2'])
a = gp.Parameter(m, 'a', domain=i, records=[['i1','1'], ['i2','2']])
z = gp.Variable(m, 'z')
eq = gp.Equation(m, name="eq")
eq[...] = gp.Sum(i, a[i]) == z
```

is a GAMSPy program (GAMS package for Python), representing the following standard GAMS syntax:

```
Set i / i1, i2 /;
Parameter a(i) / i1 1, i2 2 /;
Variable z;
Equation eq;
eq .. sum(i, a(i)) =e= z;
```

Is the GAMSPy version more verbose than standard GAMS, where there is for instance not even a single `'` or `"` in use? Sure. But with a modern text editor with hints and autocompletion, the GAMSPy notation can be written quickly, and will feel more accessible and readable to anyone with a bit of prior Python experience. GAMSPy is actually an example of an old DSL (GAMS) language repositioning itself in the Python era. Whether or not it is out of love is not known, but it surely is out of necessity.

So how does Python run this GAMSPy program? In reality, the Python program is not "run" in the sense that Python runs it normally line by line. What happens instead is that a method like `.Set()` in `gp.Set(m, 'i', records=['i1','i2'])` is not actually manipulating Python objects, but rather produces a line of GAMS code (`Set i / i1, i2 /;`) which gets executed together with the other emitted GAMS lines, when the user calls the model solver.

This kind of code-emission could also be used for Gekko, where one could imagine a Python statement like the following:

```
gk.Decomp("qBNP", eq = "e_qBNP", t = [2028, 2040], op = "d")
```

being used to emit a line of Gekko code (as a string) like the following:

```
decomp <2028 2040 d> qBNP from E_qBNP;
```

which then gets executed by Gekko. A drawback of this procedure is that the emitted Gekko code must be parsed and compiled by Gekko, so an even deeper integration would be for Python.NET to interface directly from `gk.Decomp()` into an equivalent C# method, bypassing the Gekko language syntax and parser completely. That C# method must then have a compatible signature:

```
Decomp(string name, string eq = null, int[] t = null, string op = null)
{
  ...
}
```

Like the Python method, this C# method expects a mandatory first argument followed by three optional named arguments: `eq`, `t`, and `op` (operator). Here, regarding `t`, the C# method expects an array of integers, but the method could be easily overloaded to also accept for instance an array of strings or other structures. Such interfaces between Python and C# methods are very easy to handle using Python.NET.

In addition to deeper syntax integration, another thing that could be done to integrate Gekko further with an environment like Python could be to use dataframes as data containers, instead of – or in combination with – Gekko databanks. Gekko can already write to the Apache Arrow format, and because Apache Arrow is very well integrated into dataframe packages like for instance Pandas or Polars, this format would be obvious if Gekko were to use dataframes instead of Gekko databanks for storing data. If dataframes could be easily exchanged between Gekko-package and Pandas/Polars, Gekko and Pandas/Polars could massage a given dataframe in turn, with no or very little overhead.[8]

At this point, existing Gekko-exe users could consider to port/migrate their existing Gekko-exe system into a Gekko-package equivalent. Because all of Gekko-exe would be present in Gekko-package (from using simple strings as interface to deeper integration), and because the Gekko syntax and behavior is not expected to change, this migration is solely a question of converting from one syntax to another similar syntax. In that sense, a Gekko-Python package could act as a stepping stone regarding the migration of existing systems into Python, and it would definitely be much easier, less error-prone and less time consuming than porting a large Gekko-exe system directly from Gekko syntax to for instance Python/Pandas/Polars/NumPy, which are not particularly well suited for timeseries handling.

As mentioned, automatic translators or migration tools could be offered. Gekko has previously been offering automatic translators, for instance from AREMOS to Gekko 2.4 or 3.0, or from Gekko 2.4 to Gekko 3.0. So there is a lot of existing know-how regarding how to produce such translators, and much of that know-how could be transferred into a Gekko 3.0/3.2 to Python translator.

2.3 Ad b3: Using C#.NET Core for the Gekko source code

As mentioned before, Gekko is written in C# for the .NET Framework, which is gradually being phased out and will likely be considered legacy by around 2030. Over time, it will become harder to find developers familiar with the .NET Framework. Additionally, there are benefits to using .NET Core, such as it being open-source, cross-platform (runs on Linux/Mac/ARM processors), better security, better performance, etc.

Because of this, it would in itself be advantageous to migrate Gekko-exe to .NET Core, but such a migration would also yield some positive side effects regarding Gekko-package, namely that it would open up the possibility of offering Gekko-package for Linux or Mac users, too. Regarding a full Gekko-package written in .NET Core, for Linux or Mac it would in addition need some ported graphical user interface windows, like for instance the Gekko

[8] Apache Arrow promises zero-copy interop between programs, after an Arrow file is loaded into RAM. Regarding integration with GAMS and gamY (and GAMSPy), Gekko reads/writes GAMS .gdx files.

plotting or decomposition popup windows, but the rest of Gekko ought to run without modifications on Linux or Mac. Since many larger data-science projects use Python on Linux servers, this cross-language compatibility would be advantageous.

Gekko currently runs on .NET Framework 4.6.1. The latest version in the .NET Framework is 4.8.1 (from 2022), and no further versions will be released after 4.8.1. With 6-8 years of support anticipated, support for the framework might become an issue around 2030. This not only concerns whether Gekko can technically run on a given Windows and .NET version but also how stable, performant and secure it is. When the .NET Framework becomes truly legacy, new Windows versions will no longer support it.

Upgrading from .NET Framework to .NET Core (often simply called ".NET") requires source code changes. While the fundamental C# syntax doesn't change, many built-in components change API signatures or functionality/content. Additionally, Gekko uses third-party software (dll's), and it needs to be ensured that .NET Core versions of these are available (like for instance the parser, graphing engine, Excel and GAMS interfaces, etc.). In principle, it is "just" a migration, but migrating might still demand some resources.

So to sum up regarding .NET Core, it would be entirely possible to keep running Gekko-exe on the .NET Framework for a long time to come, but still with a limited time horizon compared to .NET Core. Regarding Gekko-package using for instance Python as host language, migrating to .NET Core would make it possible to offer Gekko-package for Linux and Mac users too, provided that some Gekko popup windows are implemented in a cross-platform fashion, for instance using the cross-platform Avalonia, or a web-based interface like Plotly or similar. More about this in section 3.6.

Whether or not to migrate to .NET Core probably also depends upon how time consuming that would be, and perhaps a pilot project ought to be performed, migrating some value-added minor Gekko module to .NET Core to get a sense of how demanding the migration is in practice.

2.4 Ad b4: More ambitious Gekko-package, using another language than C#

Up to now, the Gekko source code has stayed in C#, for either .NET Framework or .NET Core. However, the most used number-crunching system languages used for packages inside for instance a Python or R host language are C, C++ or Rust, due to their speeds and versatility.

Neither C, C++ or Rust are garbage-collected memory-wise, which makes code more complicated to write than for a garbage-collected language like C# – where RAM data structures are automatically de-allocated when not in use anymore. Rust introduces some new ideas regarding memory ownership though, easing some of that memory burden compared to C/C++. Because of the absence of garbage collection, and also because of other things, C/C++/Rust programs typically run faster and use less memory than C# programs. But not an order of magnitude faster, and C# is by no means a slow language, cf. the speed tests in section 3.11.

Another advantage of using C/C++/Rust in a Gekko-package would be that it would hence be possible to precompile the Gekko source code into efficient binary executable files, for both Windows, Linux and Mac. When deploying such a Gekko-package, nothing much would need to be downloaded apart from the not too large Gekko binaries for the particular operating systems. However, it should be mentioned that the build process regarding these operating systems may be complicated, and debugging this build process on three different platforms could be a pain.

If instead the Gekko source code remains as C# and is migrated to .NET Core, the process is easier seen from the Gekko developer's point of view. Stated a bit simplified, .NET Core would compile the C# code and run it on either Windows, Linux or Mac using a .NET Core Runtime. For this to work on Linux or Mac, the user would have to install the .NET Core Runtime (if not already installed), which would be a download of around 100-200 MB. This would probably not be an absolute roadblock, but an inconvenience for sure.

If a host language like Julia was chosen, in principle all of the value-added parts of Gekko could be translated into Julia syntax proper, because Julia often runs with speeds comparable to fast languages like C/C++/Rust. A drawback to porting from C# to Julia, however, is that Julia does not support true static types (or classes/objects/class methods for that sake), cf. section 3.2 on static versus dynamic languages.

If a host language like for instance Mojo (which promises Python compatibility, but also much faster execution and optional static types and more) gains traction, translating the Gekko C# code into this static-Python generalization would make sense, but Mojo is probably at least a few years from providing a nearly full Python experience, and a Mojo-for-Windows version may lag behind.

Regarding host languages like for instance R, Julia or Mojo, see section 3.4 for a discussion of their relative advantages and disadvantages compared to Python. R is similar to Python in the sense that performant packages for R are also written in C/C++/Rust, but R is less of a general purpose language than Python, and is seen more as a niche language for statistics and data wrangling/vectorization, and it is much less used than Python. For that reason alone, choosing Python over R as a host language for Gekko would make sense.

A general drawback related to using another system language than C# for Gekko-package is that in that case, Gekko-exe and Gekko-package would diverge regarding the code base. Changes would have to be applied and tested in two different languages, as long as Gekko-exe is supported, and this would entail double work and would be painful.

3 Even more details

3.1 Types of programs and programmers

Before describing the difference between static and dynamic computer languages, we will offer a simplistic distinction between “data scientists” and “software engineers”.

- **Data scientists** may not think of themselves as programmers, and they are often focused on cleaning and wrangling data, visualizing it (often with the help of statistical methods), and perhaps even using mathematical relationships in the form of equations and models (and machine learning) to gather further insights about the data. Often, the data scientist’s programs may contain a lot of complexity regarding for instance the statistical methods used, or the mathematical relationships investigated (including machine learning aspects), but without directly using the full extent of the features of the programming language, like for instance more complicated programming concepts such as object oriented programming, functional programming, interfaces, generics, delegates, parallelism, etc. In that sense, to a software engineer’s eyes, data scientists are often seen as mostly performing “scripting”.

- **Software engineers** often use the full capabilities of the programming language, because they are more focused on the long-term viability, robustness, correctness and maintainability of the code. Software engineer’s programs tend to be large, potentially spanning millions of lines of code, and code that was written once may not be looked at again for years (unless the code fails). Encapsulation of code is encouraged, for instance putting components into well-described functions/methods. Object orientation can encapsulate data in a similar way, and a type system can make sure that only certain variable types may enter certain functions. With large code bases, as is often the case regarding software engineering, variables may stem from completely different parts of the software, and therefore a strong and preferably static type system acts as a kind of guardrail against mistakenly thinking that a variable input is, say, a numerical value, when it is in reality for instance a text string (cf. the examples in the next section).

For larger systems of source code, using a software language with dynamic types may soon end up being hard to manage. This is for instance one of the reasons that TypeScipt (with static types) was developed as an alternative to JavaScript (with dynamic types), with TypeScript gaining popularity for larger codebases used in internet browsers.

3.2 Static versus dynamic computer languages

So-called "static" computer languages are often perceived as being "strict" or "hard", whereas "dynamic" computer languages are often praised for their easy syntax and general ease of use, often in an interactive manner. In this section, we will provide a more fair description of the advantages and drawbacks of each approach.

One important characteristic regarding a computer language is its type system. People from a math or statistics backgrounds will often wonder what types are for? If I write `x1 = 1.2` or `x2 = 34`, is it not obvious that x1 and x2 are floating point numbers or integers, and if I write `x3 = ‘56’` is it not obvious that x3 is a text string? Why should I have to write code like `x1: float = 1.2`, `x2: int = 34` or `x3: string = ‘56’`? And if I want to define a function that adds two numbers, isn't it enough to write it as for instance `def Add(x1, x2): return x1 + x2`, telling the machine that `Add(... , ...)` is just kind of another way of writing `... + ...` ? Using a type-free function like that, you can use `Add(1.2, 34)` and expect to get 35.2 returned, but you can even use `Add(1.2, ‘56’)`, but what would the result be in that case? In many computer languages, `1.2 + ‘56’` will be equal to the string '1.256' because the computer language silently "promotes" 1.2 into the string '1.2' before doing a string concatenation of the strings '1.2' and '56'. If the inputs stem from an internet form accepting user input, showing such a sum as '1.256' rather than 57.2 (or raising an error) would be bad.

Computer languages that allow the above kind of type flexibility are often called "dynamic" languages, in contrast to so-called "static" languages that are more rigid regarding types. In a static language, we could have a function `Add(x1: float, x2: float)`, and the compiler will complain if you try to use `Add(1.2, ‘34’)`, because that combination of types has not been defined (in contrast, using `Add(1.2, 34)` will typically work out even though 34 is an integer, because the integer can be automatically 'promoted' to a floating point value).

But then why use dynamic programming languages at all? The answer is that the flexibility also has advantages. For smaller (possibly interactive) programs, omitting the types may help readability, because there is less clutter:

```
x1 = 1.2;
x2 = 34;
x3 = ‘56’;
x4 = Add(x1, x2);
x5 = Add(x3, x4);
```

The result of this is the string '35.256', which is not that hard to envision. It is not just about readability, though. Without types, the function `Add()` will just implicitly use the operator `+` on any two input arguments, which can be very practical if you for instance start out defining a model on 64-bit precision floating point numbers, and then want to switch to 32-bit precision input (or a mix of 32- and 64-bit precision).[9]

[9] This ability, and the ability to do it at high speed, is one of the main selling points of Julia.

So the flexibility can have advantages. But then consider code like this, perhaps from part of a large software system:

```
x1 = f1();
x2 = f2();
x3 = f3();
x4 = f4(x1, x2);
x5 = f5(x3, x4);
```

What does this do? What are the x's? Are they numerical values or strings? Or might they be more complicated objects like for instance lists, arrays, database entries, http requests, file streams, images, and so on? And what are the f()'s, what goes in and out of these? This cannot be seen from the function definitions, which may just look like `def f4(a, b): return a + b;` without any restrictions on the types of `a` and `b`. The user must then rely on comments in the code, or in a source code documentation to figure out if `f4()` is intended to add to integers, two strings, or two lists, etc.[10]

For a large complicated software system like Gekko, using a dynamically typed language would be quite infeasible. Dynamic code is great for fast prototyping, data science, machine learning etc., but for enterprise level software engineering, the source code could quickly become unmaintainable without static types, unless comments, source code documentation and unit tests are of a very high standard.

3.3 Host languages and the two-language problem

The so-called two-language problem in data science is the problem that the host language and the package language are often different. For instance, for Python and R, number crunching packages are often written in for instance C, C++ or Rust, so that data scientists write the data science scripts, and software engineers write the package programs.

An exception to this rule is the relatively new data science language/platform Julia, which offers dynamic types combined with performance not too far off C/C++/Rust for many tasks, using something called multiple dispatch, which is the killer feature of Julia that distinguishes it from other data science languages.

A software like Julia was intended to solve the two-language problem, and in that sense provide a better Python (and better R and better MATLAB). Julia 1.0 was launched 2018, but perhaps nowadays the two-language problem is not seen as quite that large of a problem, and there are also some drawbacks to the Julia language. Another example of a language/platform that aims at solving the two-language problem is the much newer Mojo project, offering to run Python with either the usual dynamic syntax (slow), or in a generalized Python syntax variant with static types (fast).

[10] With dynamic languages, type errors are caught when running (not compiling) the program, and type errors are therefore often expected to be caught by the unit testing system (which demands that the test cases are quite rigorous).

3.4 Choice of host language: Python vs. R vs. Julia vs. Mojo

Selecting **Python** as host language is clearly the least risky strategy, because Python is ubiquitous nowadays, with a vast number of users and packages, and at the moment nothing seems to block Python's upwards trend. As mentioned in section 1.1, Python has become the preferred "glue" language for AI and machine learning (including generative AI chatbots), and does anyone expect these fields to decline in the coming years? Python is the most used language for data science, too.

R is also popular for data science, with great data wrangling and visualization packages like the tidyverse and great graphing possibilities, but Python is just more of a general purpose language, whereas R has more of a vector/matrix syntax flavor. Apart from being less used than Python, using R as a host language for Gekko would make the syntax integration more difficult, because the language is less general purpose (heavy R users may disagree here). If Gekko was a software used primarily for statistics, R would clearly be a contender as host language, but Gekko is more about wrangling timeseries and analyzing model equations and similar.

One of the advantages of **Julia** is that the syntax is a bit Python-like (though still more MATLAB-like), but with a bias towards math textbook notation, like for instance starting arrays at position 1 (like R and MATLAB, too). This is in contrast to Python and fast system languages like C/C++/Rust/C#, where arrays start at position 0. In spite of having dynamic types, Julia runs fast, and so Julia packages are typically written in Julia proper. Had Julia won out against Python 10 years ago, Julia could be considered as a host language for Gekko, but the reality is that though Julia is popular, its popularity or number of packages is nothing near Python's, the growth of Julia nowadays is more linear than exponential, and Python is simply very entrenched in the domains of data science, machine learning, etc. Will Julia one day take off in a big way regarding popularity, or will it forever stay more of a niche language competing with R and MATLAB, with use cases more for specialized scientific computations, like for instance numerical simulations of physical systems? This is hard to say, but at the moment there is a feeling that Julia is stagnating a bit adoption wise.

Another thing about Julia is that static types cannot be enforced, and dynamic types are hard to work with for large projects, cf. section 3.2. Implementing for instance a weather forecasting system in Julia is probably a great experience, using Julia's differential equations solvers, etc. Variables for a weather forecasting program may be strings, timestamps, floating point numbers, vectors, matrices, arrays/tensors, even complex numbers, and differences in number precision could be handled seamlessly in Julia, too. But in software like Gekko there is inevitably a large number of different object types, for instance for user interfaces where something even as simple as a textbox inherits from a "user control" type, which inherits from "user interface element" type, which inherits from "dispatcher object" type, etc. Static types and real object orientation is not to be expected in Julia, among other things because it goes against the whole philosophy of the language.[11]

Mojo is a very new language/platform, and still too new to realistically consider as a host language for Gekko. The idea of Mojo is to accept the Python syntax as it is, admitting that it is actually quite clean and not broken in any significant way, and on top of that create a language variant (generalization) with static types. This is very similar to what TypeScript did to JavaScript, and the idea is that for some not-too-complicated parts of, say, a Mojo package, standard Python could be used, but for more speed critical "hot" parts, a static language dialect could step in seamlessly. This faster dialect would look very Python-like, but would in fact work more like a systems language like Rust, with static types and without memory garbage collection (using a memory ownership system instead). Doing simple looping etc. in the fast static-types Mojo variant would be easy enough for existing Python data scientists, whereas programming more complicated systems in the static-types Mojo variant would appeal more to software engineers and could be challenging, because it would be "close to the metal".

Mojo also promises a newer compiler architecture (MLIR) and the possibility to support GPU computations in an integrated way. It remains to be seen where Mojo goes, but for now it is not yet out of beta, and it is even not available for Windows yet (unless using Linux emulation, cf. the appendix 3.11).

3.5 GAMS and solver considerations

An important component in choosing a host language for Gekko is of course the host language ecosystem. As mentioned above, the Python ecosystem is vast, and it should be mentioned that GAMS recently has released a GAMSPy package for Python, making it possible to write model equations in a Python-like syntax, have these translated into normal GAMS form (under the hood), and then get the model solved as usual, using a solver. No doubt GAMS has had the same considerations: is the standard GAMS software viable outside of a data science host language? GAMS can also interface to for instance R, MATLAB, and also .NET, but more in form of API's for the databank .gdx format. The Python integration via GAMSPy goes far beyond that.

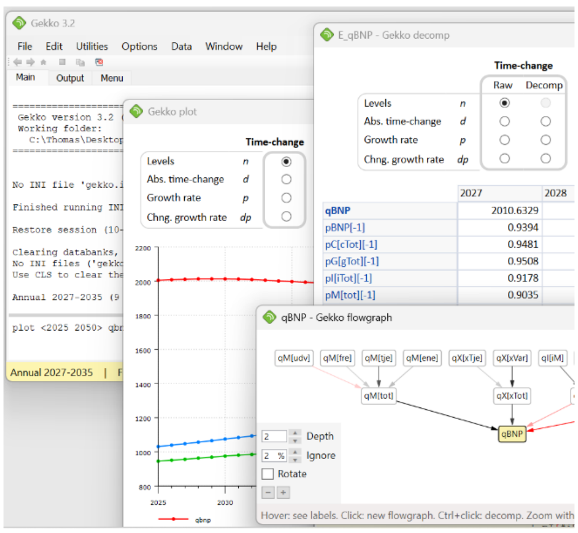
R, MATLAB and Julia have their own ways of defining and running equations and models, too, including the use of solvers also used by GAMS (for instance CONOPT), but the integration with GAMS in partiular is just not as tight regarding R, MATLAB or Julia as it is with Python and GAMSPy.[12]

[11] It should be mentioned that the DSGE model solver system Dynare was originally developed in MATLAB, but has recently been partially ported to Julia. When the Dynare main developer was asked in a Dynare forum about a port to Python, too, the answer was the following: "Actually, even though Julia is nice, we are also interested in Python, because of its ubiquity and huge potential. So in the future we may also offer a Python version". Python seems hard to ignore.

[12] Pyomo for Python should also be mentioned as a modeling language alternative to GAMSPy, and so should the modeling language JuMP for Julia. Both of these can interface with GAMS in order to use GAMS solvers, but

A Gekko-package for Python could be designed to integrate well with GAMSPy regarding for instance syntax for sets, sum loops, etc., and array-series in general. It is not that GAMS is necessarily a great language when it comes to its own syntax (it is not: the syntax feels very old), but the fact is that many Gekko users also use GAMS models to some extent, and Gekko already supports reading GAMS databanks and models, so it would probably be wise to at least take a look at the GAMSPy syntax and logic when designing a Gekko-package.

3.6 What to do about Gekko popup windows?



```
====================
 Gekko version 3.2
 Working folder:
   C:\Thomas\Desktop
====================

No INI file 'gekko.i

Finished running INI

Restore session (10-

Clearing databanks,
No INI files ('gekko
Use CLS to clear the

Annual 2027-2035 (9

plot <2025 2050> qbn
```

	2027	2028
qBNP	2010.6329	
pBNP[-1]	0.9394	
pC[cTot][-1]	0.9481	
pG[gTot][-1]	0.9508	
pI[iTot][-1]	0.9178	
pM[tot][-1]	0.9035	

A Gekko-package would need to be able to show popup windows like the plot window, the decomposition window, the flowgraph window, the trace viewer, and others. There are many possibilities regarding this, the easiest being just to use the current popup windows. This would work for Windows users, running a Gekko-Python package.

If the package needs to be able to run multi-platform, for instance on Linux or Mac, the easiest way of doing this would be to port the Gekko source code to .NET Core (which supports Linux and Mac), but then the popup windows would still be a problem in Linux/Mac, because the technology used for these windows (WPF for .NET) is not and will not be supported on non-Windows platforms. Instead, Avalonia could be used, since it is very similar to WPF and inspired by it, providing cross-platform popup windows.[13]

A more ambitious solution to the question of the Gekko popup windows would be to use a cross-platform ecosystem like Plotly/Dash, leveraging the cross-platform capabilities of modern web browsers. As an alternative, Electron+React+Flask could also be considered, but Plotly/Dash already uses React+Flask under the hood, so probably Plotly/Dash already covers almost all relevant GUI functionality.

none of them have GAMS integration as deeply as GAMSPy. In fact, Pyomo and JuMP are GAMS competitors, not GAMS users.

[13] Using Avalonia XPF would be much easier, since it supports WPF syntax directly. But XPF is a commercial product and would probably be too expensive for a project like Gekko.

3.7 Databanks and an API for Gekko databank files (.gbk)?

Gekko databank files use an open format implemented via so-called protocol buffers (developed by Google). Still, a dedicated API could be developed, for instance for Python. This could perhaps transform a Gekko databank into a corresponding dataframe, perhaps using the Apache Arrow format as a bridge.

Since Gekko is already capable of writing Apache Arrow files, an API could rather easily be made part of a Gekko-package, possibly as one of its first features. As mentioned in section 2.2, Apache Arrow currently serves as the backbone format for Python packages like Pandas 2 or Polars.

A challenge regarding dataframes and Arrow files could be metadata, though, particularly Gekko data-traces. These traces are technically a directed acyclical graph of individual traces pointing to each other, and it would take some work to integrate them inside Arrow files (these support metadata, so it ought to be possible). Otherwise, the data-traces must be omitted.

3.8 Gekko source code documentation

There have been two major source code documentation projects for Gekko, in 2014 and 2021, partially in collaboration with the IT department of Statistics Denmark (cf. this page). Better source code documentation, including a well-structured and well-commented source code, reduces the dependency on individual developers, cf. section 3.9.

If Gekko is going to be offered as a Python package, and C# is still used as the system language, it would make sense to focus on documenting the value-added parts, cf. the diagram in section 1.2, for instance the timeseries handling (including data-traces), model decomposition, etc.

One thing missing in the current source code documentation regarding Gekko-exe might be a detailed, step-by-step guide showing how to set up Gekko in Visual Studio from the GitHub repository, make a code change, and ultimately produce a new installer package (including a new Gekko.exe file) for user distribution.

The source code of Gekko varies in complexity. Many sections involve standard C# structures and loops, which any programmer with basic programming knowledge could modify. Examples of difficult parts could include the ANTLR parser integration, transforming parser AST syntax trees into internal C# code, model solving, decomposition and flowgraphs (including the handling of so-called GAMS scalar models), Statistics Denmark databank interface, Gekcel (Excel add-in), data trace functionality, WPF windows, etc. But again, as mentioned above, perhaps the value-added parts ought to be documented first.

3.9 Dependencies on individuals

With a few exceptions, Thomas Thomsen has written all of Gekko's source code, around 150,000 lines of C#, when including {}-curlies. This person dependency has partly been a question of resources. A potential generational change in terms of the developer would always be possible since Gekko is open source and available on GitHub. However, the question is how easy or difficult this would be in practice?

When developing software in the data wrangling and modeling domains, there probably is an element of "package or perish", because stand-alone applications for data science etc. run the risk of slowly dying out due to the pressure of new users expecting such capabilities to be present in the form of a package inside a host language, rather than being self-hosting in the form of an independent application.

The Gekko project will try out the package route, and in that process more collaboration is to be expected. Existing Gekko users would probably at least consider moving (parts of) their existing Gekko systems to a Gekko-package in the longer run, and in that process, questions would arise about how to use the capabilities of Gekko most naturally in a context of a language like for instance Python, and perhaps also in the context of dataframe packages like Pandas or Polars, and number crunching packages like NumPy.

The risk of relying on one person is probably not as much about further development of Gekko as it is about ensuring that Gekko (that is, at the moment: Gekko-exe) runs smoothly and securely on any given Windows version, cf. the next section.

3.10 Risk assessment and existential threats regarding Gekko-exe

As mentioned in the preface, this paper started out being about risk assessments. From an existing Gekko-exe user's perspective, there is (A1) the risk of no further development of Gekko-exe, and (A2) the additional risk of Gekko-exe becoming deprecated and not functioning at all. In Denmark, some significant government data handling systems and economic models depend upon the timely functioning of Gekko; therefore, the latter risk is not to be ignored.

If Gekko-exe were to become frozen in time (no longer receiving new features), it does not necessarily mean that existing users would abandon it, because existing users may have grown familiar with the software (have accumulated "human capital"), and migrating to something else is resource intensive and painful (and error-prone, too). Does being frozen in time imply an existential threat to Gekko and its users as a software platform? Not necessarily, at least not in the shorter run, as long as the software still installs and runs.[14]

But as long as there is at least some funding regarding Gekko-exe, the current Gekko editor and developer will make sure that any existing Gekko-exe can run on new Windows versions, and will attempt to fix any problems if this is not the case. In the absence of funding, however, or with too little funding, the Gekko editor might pursue other

[14] It will inevitably become legacy software sooner or later, though.

endeavors, and in that case, another editor (or contributor, or external consultant) would have to step in, concerning for instance security or incompatibility with new Windows versions.

To sum up, an existential threat regarding Gekko-exe would be that (B1) the current Gekko editor resigns because of either lack of funding or lack of motivation or other reasons, combined with (B2) problems running Gekko-exe in a stable and secure fashion on a given Windows version. If this were to happen, and the problems could not realistically be remedied by third-party software developers, Gekko-exe would amount to a "burning platform". But before imagining such worst case scenarios, it must be emphasized that given a reasonable amount of up-to-date source code documentation on the most critical parts of Gekko, a third-party software developer ought to be able to fix pressing bugs and problems.[15]

Whether or not it is perceived as a grave risk if Gekko-exe were to become frozen in time due to being abandoned by the editor and main developer is of course debatable. Pursuing a Gekko-package project, and providing some reasonable funding to that end, would mitigate the frozen-in-time risk, because such a Gekko-package project would not only imply a migration in the direction of host languages like Python, but would also ensure the continuing development of Gekko's capabilities, including the capabilities of Gekko-exe.

3.11 Appendix: speed tests for C#, C++, Python, Julia and Mojo

To test speed for very simple speed-critical operations in some of the languages mentioned in this paper, it has been tried out to calculate a sum of values from from 1 up to *n*, where *n* is a large number. For instance, for *n* = 100, this sum is 5050. When setting $n = 10^{10}$, on a normal laptop, this calculation takes around 5 seconds on both C#, C++, Mojo and Julia, whereas it takes about 500 seconds on Python. So, for this simple task – which is very close to the cpu "bare metal" – C#, C++, Mojo and Julia all run similarly fast, whereas Python is critically slow. (Regarding the timings, it should be noted that Mojo in these experiments runs on Windows via Linux emulation, but for a task like this, the emulation loss is likely to be small). It should be mentioned that the benchmark is very simple, and that a more realistic benchmark would involve timeseries (vector) handling, dataframe handling, file access, etc.

[15] If the pressing problem is that the .NET Framework does not work anymore or has become officially deprecated, the problem is of a different magnitude.

C#

```
public static double Sum_numbers()
{
    double total = 0.0;
    double i = 1.0;
    while (i <= 1e10)
    {
        total += i;
        i++;
    }
    return total;
}
```

C++

```
double Sum_numbers()
{
    double total = 0.0;
    double i = 1.0;
    while (i <= 1e10)
    {
        total += i;
        i++;
    }
    return total;
}
```

Mojo

```
fn Sum_numbers() -> Float64:
    var total: Float64 = 0.0
    var i: Float64 = 1.0
    while i <= 1e10:
        total += i
        i += 1.0
    return total
```

Julia

```
function Sum_numbers()::Float64
    total::Float64 = 0.0
    i::Float64 = 1.0
    while i <= 1e10
        total += i
        i += 1.0
    end
    return total
end
```

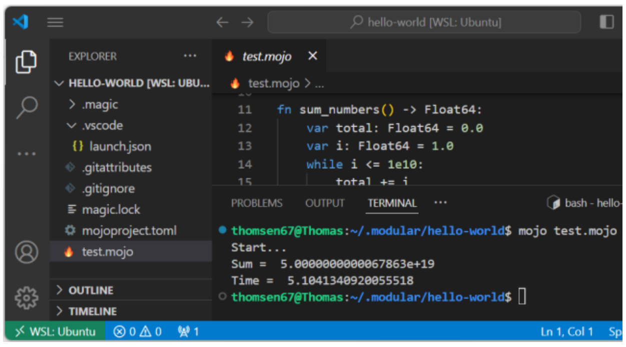
Python

```
def Sum_numbers():
    total = 0.0
    i = 1.0
    while i <= 1e10:
        total += i
        i += 1.0
    return total
```

As a side note, if the standard Python code (with “def”) is run from inside Mojo, it actually runs at the same speed as Mojo code (with “fn”), even though the Python code has no type decorations. This is probably because the compiler can infer the types, but it is still a nice speedup for completely normal-looking Python code.

With Mojo, this example can undoubtedly be sped up using SIMD instructions, parallelizing, or calling the gpu (at the cost of more complex code). However, the scope of this test is just to ensure that Mojo does not run slower than C#, C++ or Julia out of the box, for simple things like loops.

On a Windows pc, for the time being Mojo needs to run in a WSL environment (Windows Subsystem for Linux), that is, with Linux virtualization. It must be emphasized WSL is not using a separate virtual machine or dual booting, but a quite seamless component that is available for Windows 10 or 11 and often comes pre-installed (also, the so-called “Virtual Machine Platform” needs to be activated on Windows, which it already is on Windows 11). As shown in the bottom left corner of the following VS Code screenshot, WSL is used.



```
fn sum_numbers() -> Float64:
    var total: Float64 = 0.0
    var i: Float64 = 1.0
    while i <= 1e10:
        total += i
```

```
thomsen67@Thomas:~/.modular/hello-world$ mojo test.mojo
Start...
Sum =  5.0000000000067863e+19
Time =  5.1041340920055518
thomsen67@Thomas:~/.modular/hello-world$
```

Running Mojo directly as “Windows native” Python environment would of course be preferable on Windows, both from a convenience standpoint, but also because Linux virtualization on Windows entails a bit of cpu overhead. But at the moment, only Linux and Mac OS are supported. This has to do with the fact that one of the purposes of Mojo is being able to run machine learning tasks fast (using MLIR, gpu’s etc.), and such tasks are often run on Linux servers for cost reasons, and Mojo needs to monetize that first. Since Mac OS is very similar to Linux, it is easier to support than Windows.

Rumor has that the Mojo developers are contemplating a Windows version, but there is no timeline, roadmap or guarantee regarding this.[16] However, it is the most requested feature by users, and a Windows version ought to be expected sooner or later, particularly if Mojo becomes popular. Until that, Linux virtualization works quite smoothly.

There is also the question of when Mojo 1.0 is to be expected? The CEO of Mojo has recently said that something like object/classes for Mojo will probably not be finalized before the end of 2026, so Mojo 1.0 could be years off. This does not mean that Mojo cannot be used before a stable version 1.0, though.

[16] An official Mojo developer comment (summer 2024): ”... we understand how important the native Windows experience is for a large percentage of the world. The reason we haven't done this yet is purely due to limited engineering hours, and we need to focus on priorities such as GPU support to create a compelling product. We care about Windows, as you can see when we add stdlib functionality we include native Windows compatible code and tests. When we have the engineering hours available, we'll do it.”