1 The `CalculusWithJulia` package

To run the commands in these notes, some external packages must be installed and loaded.

All that is needed is to install the `CalculusWithJulia` package with:

```
] add CalculusWithJulia
```

This only needs to be done once.

However, for each new `Julia` session, the package must be *loaded* with the following command:

```
using CalculusWithJulia
```

```
using CalculusWithJulia.WeaveSupport
```

That is all. The rest of this page just provides some details for the interested reader.

1.1 The package concept

The `Julia` language provides the building blocks for the wider `Julia` ecosystem that enhance and extend the language's applicability.

`Julia` is extended through "packages." Some of these, such as packages for certain math constants and some linear algebra operations, are part of all `Julia` installations and must simple by loaded to be used. Others, such as packages for finding integrals or (automatic) derivatives are provided by uses and must first be *installed* before being used.

1.1.1 Package installation

Package installation is straightforward, as `Julia` has a package, `Pkg`, that facilitates this. The command line and `IJulia` provide access to the function in `Pkg` through the escape command `]`. For example, to find the status of all currently installed packages, the following command can be executed:

```
] status
```

External packages are *typically* installed from GitHub and if they are regisered, installation is as easy as call `add`:

```
] add QuadGK
```

That command will consult `Julia`'s general regisrty for the location of the `QuadGK` package, use this location to download the necessary files, if necessary will build or install dependencies, and then make the package available for use.

For these notes, when the `CalculusWithJulia` package is installed it will also install all the other packages that are needed.

> Installing the `CalculusWithJulia` package is the only package necessary to install for these notes.

See Pkg for more details, such as how to update the set of available packages.

1.1.2 Using a package

The features of an installed package are not available until the package is brought into the current session. A package need only be *installed* once, but must be loaded each session.

To load a package, the `using` keyword is provided:

```
using QuadGK
```

The above command will make available all *exported* function names from the `QuadGK` package so they can be directly used, as in:

```
quadgk(sin, 0, pi)
```

```
(2.0, 1.7905676941154525e-12)
```

(A command to find an integral of $f(x) = \sin(x)$ over $[0, \pi]$.)

1.1.3 Package details

When a package if *first* loaded after installation, or some other change, it will go through a *pre-compilation* process. Depending on the package size, this can take a moment to several seconds. This won't happen the second time a package is loaded.

However, subsequent times a package is loaded some further compilation is done, so it can still take some time for a package to load. Mostly this is not noticeable, though with the plotting package used in these notes, it is.

When a package is loaded, all of its dependent packages are also loaded, but their functions are not immediately available to the user.

In *typical* `Julia` usage, each needed package is loaded on demand. This is faster and also keeps the namespace (the collection of variable and function names) smaller to avoid collisions. However, for these notes, the package `CalculusWithJulia` will load all the packages needed for the entire set of notes, not just the current section. This is to make it *easier* for the *beginning* user.

One issue with loading several packages is the possibility that more than one will export a function with the same name, causing a collision.

The `Julia` language is designed around have several "generic" functions each with many different methods depending on their usage. This design allows many different implementations for operations such as addition or multiplication yet the user only needs to call one

function name. Packages can easily extend these generic functions by providing their own methods for their own new types of data. For example, `SymPy`, which adds symbolic math features to `Julia` (using a Python package) extends both `+` and `*` for use with symbolic objects.

This design works great when the "generic" usage matches the package authors needs, but there are two common issues that arise:

- The extension of a generic is for a type defined outside the author's package. This is known as "type piracy" and is frowned on, as it can lead to subtle errors. The `CalculusWithJulia` package practices this for one case: using `'` to indicate derivatives for `Function` objects.
- The generic function concept is not part of base `Julia`. An example might be the `solve` function. This name has a well-defined mathematical usage (e.g., "solve for x."), but the generic concept is not part of base `Julia`. It is part of `SymPy` and it is part of `DifferentialEquations`. That is, both package export this function name. For the user, if *both* packages are loaded, then they user **must** *qualify* which package's `solve` function they mean, as in `SymPy.solve` or `DifferentialEquations.solve`. (`DifferentialEquations` is not part of `CalculusWithJulia` and is just briefly used in these notes, though is an incredible set of packages and a testament to the strengths and power of `Julia`.)