World and Science

a Master's Thesis by John E. Niclasen

May 1, 2016

Contents

List of Figures

List of Tables

1 Introduction

1.1 A Bit of History

I have a background in programming. For eighteen years, I worked professionally as a system developer producing software and programming standards for different companies as well as my own developments. I had a basic education in programming, and I supplemented it with another higher level IT (Information Technology) education along the way.

Then in the year 2006, I decided to follow an old dream of studying physics and astronomy at the Niels Bohr Institute. With my background understanding of programming method, the usefulness of good standards, and the crucial need to *keep it simple* (the K.I.S.S. principle), I was horrified at what I saw at the university.

As so much science involve computers and data processing in this new millennium, I saw the need for better tools directed at scientists.

1.2 The World Programming Language

In the year 2009, doing my last bachelor year in physics and astronomy at the Niels Bohr Institute, I started development of a new programming language directed at science, and with scientists as the primary target group of users. As much as I could see others would benefit from better tools, I desperately needed better tools myself.

Among the problems with existing programming languages, as seen from a scientist viewpoint, is, that all kinds of technical issues are in the way to achieve ones goal. The first issue noticed, when a scientist needs to program something, is the syntax of the chosen programming language. A scientist is used to read and write mathematical symbols on a blackboard or in a scientific paper and think in terms of simple pseudo code, when dealing with an algorithm. But most programming languages require a lot of extra syntax, before the computer can understand, what is written.

An example could be calling **cosine** with π as the argument, which in many languages would be something like

```
cos(pi);
```

The extra unneeded syntax (the parenthesis and the semicolon) is just extra work and should be avoided. This is crucial! The problems start with this basic observation. A scientist (and actually all programmers) should be able to just write

```
cos pi
```

, and the computer should then understand, what to do. If we take the next (a bit more complicated) example, we can see how the syntax of many languages leads to bloated code, often lack of readability and therefore lots of future problems with maintaining code written earlier, or re-use of code written earlier. Let us raise Euler's number e to the power of π. In some languages it would be

```
pow(e, pi);
```

If solving this task using a function (**pow** in this case, which one would maybe call **power** for better readability), it should be enough to write

```
power e pi
```

Solving the task with an operator, it could be

```
e ** pi
```

, or if one chooses to use the symbol (^) instead of (**), it could be

```
e ^ pi
```

After finishing my bachelor in the winter 2009-2010, I used the next four and a half year developing a new programming language from scratch, the World Programming Language, which should solve these syntax issues as well as many other issues, so it could be a joy also as a scientist to program computers.

Syntax is one issue, datatypes another. Most programming languages have very few basic datatypes, like *integer*, *float*, *character*, *string*, etc. Some other datatypes like a file on disc, or new datatypes, like a URL or an e-mail address, are then just dealt with using typically a string. The problem with this is, that the language then needs to define different functions, for example to **read** data from a disc file or to **read** data from a file-server using a URL. Having more datatypes leads to fewer functions (which is more simple to remember), because a function can then do different things depending on the type of argument giving to it. For example one and the same **read** function could be used to read data in both of the two cases. This is a huge benefit, which can maybe only fully be appreciated, when having written code in this language for some time.

The World Programming Language has at the time of writing more than 50 different datatypes defined and more than 10 typesets (which can be seen as categorization of datatypes). See Figure 1.

```
any-type!
    none!
    unset!
    logic!
    scalar!
        number!
            integer!
            real!
            percent!
        complex!
        char!
        pair!
        range!
        tuple!
        vector!
        time!
    date!
    image!
    series!
        any-string!
            string!
            binary!
            file!
            email!
            url!
            tag!
        any-block!
            block!
            any-paren!
                paren!
            any-path!
                path!
                set-path!
                get-path!
                lit-path!
        list!
    lit-string!
    map!
    datatype!
    typeset!
    bitset!
    any-word!
        word!
        set-word!
        get-word!
        lit-word!
        issue!
        refinement!
    any-function!
        operator!
        function!
        routine!
        callback!
        task!
    any-object!
        context!
        error!
        port!
    task-id!
    node!
    handle!
    struct!
    library!
    comment!
    KWATZ!
```

Figure 1: Hierarchy of datatypes and typesets.

World defines 86 native functions and 170+ *mezzanines* at time of writing. *Mezzanines* are functions defined in the language itself, and they are in use indistinguishable from native functions. The list is too long to present here, but it can be found in Appendix A. The language doesn't have the concept of *keywords* known from many other languages. World also defines 17 native operators and a handful mezzanine operators, which are both listed together in Table 1.

Table 1: World operators.

Operator	**Description**
`**`	First number raised to the second.
`*`	First value multiplied by the second.
`+`	Add two values.
`-`	Second value subtracted from the first.
`//`	Remainder of first value divided by second.
`/`	First value divided by the second.
<	True if the first value is less than the se...
<<	Bitwise shift left.
<=	True if the first value is less than or equ...
<>	True if the values are not equal.
`=`	True if the values are equal.
`==`	True if the values are equal and of the sam...
`=?`	True if the values are identical.
>	True if the first value is greater than the...
>=	True if the first value is greater than or ...
>>	Bitwise shift right.
`after`	The place after a value in a series.
`and`	First value ANDed with the second.
`before`	The place at a value in a series.
`from`	Find a value in a series.
`is`	Check if value is series
`of`	Value at the specified position in a compon...
`or`	First value ORed with the second.
`xor`	First value XORed with the second.

It is possible for a user to expand on these functions and operators, and new user-defined functions and operators are used just like the natives.

The World Programming Language is written in ANSI C to make it as portable as possible. At time of writing, World is released in a version for Mac OS X (64-bit), Linux (64- and 32-bit), and Windows (64- and 32-bit). The executable is around 1MB (one megabyte), and it is very easy to download and install. The implementation is careful considered and continuously evaluated to make sure, it is future proof. With new mobile devices, the IT industry has undergone huge changes in the last decades. I want to be sure, World can be used in the future too, whatever that will bring us of devices and technological wonders.

World is mainly based on ideas found in the Rebol programming language created by Carl Sassenrath, which again is influenced by languages such as Self,

Forth, Lisp and Logo. World is different in, that it is geared towards science, so it has higher level math and datatypes like complex numbers, vectors, etc., which is not found in Rebol. World is also influenced by Lua and Stackless Python.

World has much in common with a human language in written form, more than most other programming languages. This means the language is a sequence of values recognized as being of different datatypes - like words, numbers, dates, URLs, parentheses, and many other - with a minimum of syntax.

World has a homepage at: http://www.world-lang.org

World can be downloaded from: https://github.com/Geomol/World

1.3 This Thesis

In this thesis, I first in section 2 describe some common programming tasks, we often face in the scientific community. In section 3, I will investigate some of the methods available to fulfil these programming tasks. Then follow a palette of examples, where I have used World to solve many different tasks all related to science. These examples should be seen as a proof of concept. It is to illustrate, how easy some tasks can be solved with the right tool. At the end I look at expanding World, point out some of the areas, World could be used in the future, and ending with a conclusion.

It is my hope, that the reader with this thesis will get some insight into the areas, that would benefit from a tool like the World Programming Language, both related to science, but also in a broader perspective.

Complexity is our enemy! Having the right tool can be the difference between success and failure.

As Frédéric Chopin is quoted:

> *Simplicity is the final achievement. After one has played a vast quantity of notes and more notes, it is simplicity that emerges as the crowning reward of art.*

Something along these lines can be said about programming, which has become such a big part of science. Programmers often refer to the K.I.S.S. principle as first described by Kelly Johnson, lead engineer at the Lockheed Skunk Works (creators of the Lockheed U-2 and SR-71 Blackbird spy planes, among many others). But few seem to really be focused on keeping things simple, and it is maybe often believed, that keeping things simple means less advanced tasks can be achieved. This is a misunderstanding and not true. A LEGO brick is very simple. Look at what advanced creations, kids can make with such simple building blocks.

2 Programming Tasks

To get some insight into the areas, where programming tools are involved in scientific work inside or outside universities around the world, let us divide the scientific workflow into three topics.

1. Collecting data
2. Research
3. Publish

An experimental scientist at e.g. CERN collects data from the experiments running in the large particle accelerator, a biologist might go out into nature and collect plants, and an astronomer collects data using telescopes. A pure theoretical scientist might only collect data by reading other scientists published work.

Much of the time, the scientist is doing research, that is process all the input from the collected data, thinking, and doing calculations.

Then at some point, the scientist will publish, what has been discovered. Publishing is both in scientific journals, but also giving talks at conferences, creating educational material for students, maybe writing internal notes, etc.

All of these areas of work can - and nowadays often do - involve programming tasks needed to be done by the scientist or co-workers.

2.1 Collecting Data

Much of the data collection is in modern science done by computers or involve computers. A modern telescope used by astronomers typically involve collecting photons on a CCD chip connected to a computer. The data collected in this way ends up on some storage device like a hard disk.

The astronomer is now faced with the challenge of getting the data from the telescope to his or her own computer or a file-server at the university, which is possible to get access to. It is necessary to have the data easily available to be able to do research using the data.

Programming scripts and small compiled programs are used in both collecting the data and in transferring the data from one computer to the next. Much of this programming is done by the astronomers themselves.

2.2 Research

Doing the research, the astronomer is faced with several different tasks involving computers and often programming. Astronomical data needs to be reduced. If the data is image data, it needs to be processed into a final image viewable on a computer screen or able to be printed out.

Often research involves heavy calculations done with the help of computers. In many situations, there isn't a function available just to be called, so the scientist write their own functions and own programs to do the calculations.

In some cases, it is required to be able to access databases with information needed in the research. Such databases can reside locally on the astronomers own computer, or it can be a large database on some external computer. If plenty and often access is needed to the data, the astronomer might choose

to write scripts and programs to ease the task of getting the data from the database.

2.3 Publish

Publishing research results involve plotting graphs. It might involve making drawings of e.g. instrument setup. Writing scientific papers is often done in TEX (or LATEX), which can be considered programming. Doing a presentation involve producing slides to be shown using a projector.

All of these kinds of publishing in modern science involve computers and often programming. In some cases, where programming isn't used today, it could be a benefit to use programming of small scripts to ease the task - to ease the workflow.

In the next section, I will look at some of the (programming) methods used today in more detail.

3 Methods

To solve the programming tasks mentioned in the previous section, and tasks in general involving computers, I will look into some of the methods normally in use today. I will use the same division of workflow.

3.1 Collecting Data

The challenge of getting data from the telescope to the astronomers own computer or a file-server at the university is being solved using different methods. If the amount of data is small, a normal e-mail can be sent with the data attached. Often the amount of data is large, and some other method is needed. The data might be available from a FTP-server (File Transfer Protocol), and the astronomer can then get the data using a **ftp** tool in the terminal, or maybe choose a solution with a graphical interface like FileZilla. The data could be placed in a way, so a web-browser can be used to get the data.

Data is in recent years also being distributed using web-services like Dropbox. Problem with such solutions is, that they involve third party companies. If a larger workflow was implemented using such service, and that company closed, changed the service or was sold to some other company, ones solution might not work from one day to the next. One might argue, that as a service like Dropbox has been around for some years now (since June 2007), and that the company might go well, such solution is pretty safe. That is not the case in the IT industry. Large and popular solutions does change ownership, as was the case with WhatsApp Messenger, a popular messaging application, which was sold to Facebook for $19 billion after a few years of business.

Solutions with control of both ends of the data transfer, and no stop on the way, means larger probability of success.

3.2 Research

Once the astronomer has got hold of the data, getting an overview of what is actually included in the data is the first challenge. Data from one observational session often contain dozen of for example FITS files. One method to figure out, what data each FITS file holds, is to load the FITS file into a viewing tool, where also the FITS header information can be seen. Sometimes the observatory wrote the wrong information into the header, and then the astronomer must investigate further and use experience to figure out, what is what.

The astronomer might then want to put FITS files of certain types into subfolders with other FITS files of the same type. That can be done in a terminal using shell commands, or it can be done with a file tool in the graphical interface of the operating system in use.

It is possible to write scripts to ease this workflow as shall be seen in the next section of this thesis.

After getting an overview, next step could be producing a final image from all the different FITS files. This can be done for example using IRAF (Image Reduction and Analysis Facility), a huge collection of tools and commands. Often the astronomer wants to get an initial view of the image to judge, if the observation went ok, and then it can be a struggle to deal with a large software package like IRAF.

Doing the research, calculations are often carried out, and the astronomer often program their own programs to do certain calculations. Sometimes it is heavy calculations requiring a supercomputer with parallel computing capabilities. This means more programming - and maybe in another language than used on ones personal computer.

To get server access can involve setup of an X-terminal session to login on the server. Some would program a shell script to ease this task.

3.3 Publish

For publishing it is required to be able to produce plots with graphs representing the scientific data. There are many tools available, and they are often large software packages with an uncountable amount of features easy to get lost in. It would be nice now and then with an easy tool, that just do the job. Some do plots in programming languages or what we could call programming environments with main goal different from plotting. Others use tools dedicated to create plots, but these tools often include some kind of programming language. It is an old question for programmers, which language is best suited for some programming task. There are good reasons, why there are so many very different programming languages. Some languages are good for certain tasks, others for other tasks.

To create graphical drawing, one could use a dedicated drawing application. Often the scientist has certain needs not found in common drawing applications, so there is room for dedicated tools. One is for example GeoGebra, that I have used myself. Some might use the drawing capabilities found in Office programs suites like OpenOffice. Office programs also often include some sort of presentation application used to make slides with. But they are huge applications, and you might not find your preferred one on your new tablet.

Then there is the TEX producing software to write scientific papers in. There exist different solutions, but many ends up writing the .tex document themselves with a simple text editor. Different tools are then used to produce a printable document (in PDF or PostScript) from that.

Many of the methods used as described above could benefit from better and easier to use tools, than what is common available. There is room for improvement.

4 A Project in Astrophysics incl. FITS Files

I was presented a common task for astronomers dealing with FITS files. Flexible Image Transport System (FITS) is a standard defining a digital file format useful for storage, transmission and processing of scientific and other images. FITS is the most commonly used digital file format in astronomy.

A typical astronomical observation produces a number of FITS files, each with different information related to the observation. There is the scientific frame (image) or frames, if for example more than one filter was used, and there are correction frames (known as BIAS and FLAT frames). Some observations also include spectrum frames. The astronomer receive all these FITS files often in one and the same file folder, even if different files hold different information and should be handled accordingly. The astronomer now face at least two tasks:

1. Get an overview of all the different files and sort them according to type (BIAS, FLAT, SCI, SPECTRUM, ...).

2. Produce a viewable image through some processing of the data in the files.

4.1 Loading FITS Files

To solve the first of these tasks in World, I first needed a way to load FITS files from within the World language. One way could be to define a routine and call an external library, that could read the file format (if one such easily available dynamic link library is found), but I choose to write a function in World, that would be able to load FITS files.

It turned out, that the full World program code to load FITS files, as well as a function to call a viewing tool named **ds9** is less than 200 lines. The full source can be found in Appendix B, and it also includes conversion of some common FITS keywords to more human readable form using a conversion table implemented as a map! type (the map! datatype in World is used for fast mapping of values, and it is implemented using a mechanism known as *hashing*).

The following example shows the use of the **load-fits** function directly in the World prompt within a terminal window. It is to show the simplicity of using such extension to the language. First the FITS extension is included, a FITS file named "m51.fits" in the current directory is loaded, and finally some information from the FITS header is printed in the terminal.

```
w> include astro/fits
w> file: load-fits %m51.fits
w> print [file/Date file/RA file/DEC file/Object]
4-Sep-2014/13:56:36 13:29:24.00 47:15:34.00 m51 B 600
```

4.2 Sort and Validate FITS Files

The next step was to sort the FITS files in sub-folders according to type - taking into account, that sometimes files get wrong attributes set in the headers by the observatory, so a semi-intelligent judgement according to rules agreed upon with the astronomer was implemented. The World source for this task turned out to be a little more than 300 lines of code, an affordable task for one programmer. The output of this step was twofold:

1. The FITS files got sorted into sub-folders.

2. A small result report in the form of a *flat* text file was presented to the user on the computer monitor.

The text report holds columns of information, each column explained by one of the header words: file, type, IMAGETYP, ALAPRTNM, ALGRNM, TCSTGT, OBJECT, EXPTIME, status, and folder. Some of the more cryptical words here are known to the skilled astronomer and recognized as header information from the FITS file. This in all gives a good overview of all the data within a number of FITS files related to one astronomical observation. An example of the output is shown in Figure 2. (Some lines were removed in the editing, because the report was too long to fit on one page.)

This was an example of using World to create a very useful tool to go into the astronomers toolbox, where it will always be at hand without too much struggle.

file	type	IMAGETYP	ALAPRTNM	ALGRNM	TCSTGT	OBJECT	EXPTIME	status	folder
ALxh010001.fits	bias	BIAS	Slit_1.0	Grism_#4	-	alfosc-calibs bias	0.002	OK	bias/1x1/
ALxh010002.fits	bias	BIAS	Slit_1.0	Grism_#4	-	alfosc-calibs bias	0.002	OK	bias/1x1/
ALxh010003.fits	bias	BIAS	Slit_1.0	Grism_#4	-	alfosc-calibs bias	0.002	OK	bias/1x1/
ALxh010004.fits	bias	BIAS	Slit_1.0	Grism_#4	-	alfosc-calibs bias	0.002	OK	bias/1x1/
.									
.									
.									
ALxh010033.fits	bias	BIAS	Slit_1.0	Grism_#4	-	alfosc-calibs bias	0.002	OK	bias/1x1/
ALxh010034.fits	bias	BIAS	Slit_1.0	Grism_#4	-	alfosc-calibs bias	0.002	OK	bias/1x1/
ALxh010035.fits	bias	BIAS	Slit_1.0	Grism_#4	-	alfosc-calibs bias	0.002	OK	bias/1x1/
ALxh010121.fits	flat	FLAT	Open	Open_(yot)	Blank17+34	FLAT 1	1.0	OK	flat/ALFLT_B_Bes_440_100_1x1/
ALxh010122.fits	flat	FLAT	Open	Open_(yot)	Blank17+34	FLAT 2	1.07	OK	flat/ALFLT_B_Bes_440_100_1x1/
.									
.									
.									
ALxh030125.fits	flat	FLAT	Open	Open_(yot)	Blank17+66	FLAT 1	69.2	OK	flat/ALFLT_i_int_797_157_1x1/
ALxh030126.fits	flat	FLAT	Open	Open_(yot)	Blank17+66	FLAT 2	93.74	OK	flat/ALFLT_i_int_797_157_1x1/
ALxh030127.fits	flat	FLAT	Open	Open_(yot)	Blank17+66	FLAT 3	138.08	OK	flat/ALFLT_i_int_797_157_1x1/
ALxh030138.fits	sci	-	Open	Open_(yot)	m51_1	M51	60.0	OK	sci/ALFLT_V_Bes_530_80_1x1/
ALxh030139.fits	sci	-	Open	Open_(yot)	m51_1	M51	100.0	OK	sci/ALFLT_B_Bes_440_100_1x1/
.									
.									
.									
ALxh030196.fits	sci	-	Open	Open_(yot)	m17	m16	60.0	OK	sci/ALFLT_i_int_792_157_1x1/

Figure 2: Sort FITS output report example.

5 Parallel Computing

In some scientific applications, heavy calculations are carried out by supercomputers. To shorten the time to run such calculations - it could be simulations of some scientific model - calculations are done in parallel; what is known as *parallel computing*. Calculations are done with vectors of numbers using vector processors, which are CPUs that can execute the same instruction on large sets of data.

Modern GPUs (Graphics Processing Unit) on the graphics card in our modern computer (or sometimes directly located on the motherboard beside the CPU) are co-processors that have been heavily optimized for computer graphics processing. Computer graphics processing is a field dominated by data parallel operations - particularly linear algebra matrix operations.

There are several software solutions to do general purpose computation on GPUs. One such solution is known as *Bohrium*, which is being developed by the eScience group at the Niels Bohr Institute.

5.1 Bohrium

Bohrium is an API (Application Programming Interface), and it consists of a large number of functions, which are accessible through a dynamic linked library. In World, functions in such libraries are called routines (of type routine!), and to be able to call the routines, two things are needed:

1. The library needs to be loaded with the **load-library** function
2. Routines needs to be defined to call the library functions

The following World code define a word, **libbhc**, and sets the word to the result of calling **load-library**. **load-library** takes one argument, which is the disk file, that is the actual dynamic linked library, in this case named "libbhc.so".

```
libbhc: load-library %/home/john/.local/lib/libbhc.so
```

The Bohrium dynamic library, "libbhc.so", is a 4.6MB file on disk.

After having a reference to the library, routines can be defined using this reference word. The Bohrium function, **bh_multi_array_float64_add**, is used to add two vectors of 64-bit floating point numbers together. The function takes three arguments: **out**, which is where the result of the addition should end up, and the two vectors to be added, **lhs** and **rhs**. The function has no return value. The World code to define a routine, **bh-real-add**, is shown below.

```
bh-real-add: routine [
    "Addition."
    [typecheck]
    libbhc "bh_multi_array_float64_add" [
        out [handle!] pointer   "Output"
        lhs [handle!] pointer   "Left hand side"
        rhs [handle!] pointer   "Right hand side"
    ]
    void
]
```

As is seen, the code is quite readable. The strings within the definition is used with the World **help** function. Having defined the routine as above, it is now possible to get help for this function at the World prompt:

```
w> help bh-real-add
Usage:
    bh-real-add out lhs rhs

Description:
    Addition.
    bh-real-add is a routine!

Arguments:
    out -- Output [handle!]
    lhs -- Left hand side [handle!]
    rhs -- Right hand side [handle!]

Special attributes:
    typecheck
```

To be able to add two vectors together using this routine require the Bohrium workflow to be followed. The vectors has to be defined for Bohrium, and to access the result, sync needs to be activated. Using Bohrium, the user do these things and then tell Bohrium to execute all instructions with a **flush** call.

To figure out how to define all the Bohrium functions to be used as routines within World, I looked at the C header file, **bh_c.h**, which comes with Bohrium.

C function prototypes

The function prototypes in the **bh_c.h** header file is divided into categories as listed in Figure 3. There are more than 1600 functions defined in Bohrium, so it is a rather large library. Many functions come in 8-, 16-, 32-, and 64-bit versions, both signed and unsigned, and both integer, float and complex numbers. There are functions operating on 8-bit booleans. There are functions with lhs (left-hand-side) and rhs (right-hand-side) variations. Many if not most functions would never be used in an application targeted at astronomy, and I would not recommend support for all functions as routines in a World interface for Bohrium. It would be counter-productive to do that. It makes much more sense to support a selected few functions, and if new functionality is required, it is a small job to add support for new routines in World, as has been shown above. It is a benefit using dynamic linked libraries, that it isn't necessary to support all the functions within a library. We have the benefit to pick and choose only the functions, that are required for an application or a field of science and leave the rest for a possible future use.

- Common runtime methods
  - Forward definitions
    - bh_multi_array_float64_new_empty
    - bh_multi_array_float64_set_data
    - bh_multi_array_float64_sync
    - bh_multi_array_float64_destroy
    - etc.
- Copy methods
  - e.g. bh_multi_array_float64_identity_float64
- Binary functions
  - e.g. bh_multi_array_float64_add
- Unary functions
  - e.g. bh_multi_array_float64_cos
- Reduction functions
  - e.g. bh_multi_array_float64_add_reduce
- Accumulate functions
  - e.g. bh_multi_array_float64_multiply_accumulate

Figure 3: Bohrium function categories.

5.2 helloworld Test

The first thing recommended by the Bohrium team is to get their *helloworld* test running, and that I did. The test example, "helloworld.c", is a 64 line C source running some basic functionality of Bohrium. I ported this C source to World, and the World source can be seen in Figure 4.

```
World [
    Title:  "Bohrium test"
    Author: "John Niclasen"
]

include bohrium

shape: vector [sint64 2 [3 3]]
stride: vector [sint64 2 [3 1]]

; Sequence of ones
a: bh-real-new-ones 2 shape

; Range from 0 - 9
r-flat: bh-integer-new-range 9

; Reshaped to 3x3
r-shaped: bh-integer-new-view r-flat 2 0 shape stride

; Make into reals
b: bh-real-new-empty 2 shape
bh-real-identity-integer b r-shaped

; Do actual computation
output: bh-real-new-empty 2 shape
print "adding"
bh-real-add output a b

; Issue a sync instruction to ensure data is present in local
    memory space
print "sync"
bh-real-sync output

; Execute all pending instructions, including the sync
    command
print "flush"
bh-runtime-flush

; Grab the result data
print "get data"
data: as [vector! double 9] bh-real-get-data output

; Print out the result
print "Adding ones to range in 2D:"
repeat i shape/1 [
    repeat j shape/2 [
        prin [pick data (i - 1 * stride/1) + (j - 1 * stride
            /2) + 1 ""]
    ]
    prin newline
]

; Clean up anything that was allocated
print "destroying"
bh-real-destroy a
bh-real-destroy b
bh-real-destroy output
bh-integer-destroy r-flat
bh-integer-destroy r-shaped
```

Figure 4: Bohrium helloworld test.

The Bohrium routines defined in World used in this example is listen in Table 2. The full World source needed to define these routines to be able to run the "helloworld" example is listed in Appendix C.

Table 2: World routines from Bohrium helloworld example.

Routine	Description
bh-real-new-ones	Construct a new one-filled array.
bh-integer-new-range	Construct a new array with sequential numbers.
bh-integer-new-view	Construct a new array from an existing view.
bh-real-new-empty	Construct a new empty array of 64 bit floats.
bh-real-identity-integer	Make into floats.
bh-real-add	Addition.
bh-real-sync	Sync the current view.
bh-runtime-flush	Execute all pending instructions.
bh-real-get-data	Gets the data pointer from a view.
bh-real-destroy	Destroy the array and release resources.
bh-integer-destroy	Destroy the array and release resources.

5.3 Minimal Add Test

To illustrate what is required to do any computation with Bohrium, I created a minimal addition test, which is listed below incl. comments.

```
include bohrium

v: make vector! [sint64 2]
v/1: v/2: 3     ; Putting some values into the vector

a: bh-real-new-empty 2 v    ; Create Bohrium vector from v
b: bh-real-new-empty 2 v
output: bh-real-new-empty 2 v   ; Create output vector
bh-real-add output a b          ; Add a and b giving output
bh-real-sync output             ; Sync memory
bh-runtime-flush                ; Flush commands

data: bh-real-get-data output   ; Get output data

bh-real-destroy a               ; Destroy Bohrium vectors
bh-real-destroy b
bh-real-destroy output
```

5.4 Discussion

The idea behind Bohrium is really good. Write once and have your heavy calculations carried out by the GPU on your graphics card, or move your source code to a supercomputer with vector CPUs, compile it, and get the speed increase there.

I was never able to run Bohrium on my MacBook Pro under Mac OS X. It also was not possible with the help of the Bohrium team. I did my tests with the Bohrium Ubuntu package. Ubuntu is a Linux operating system, and to use it, I installed it under OS X using VirtualBox, which is software making it possible to run several operating systems.

The Bohrium API is not as simple as one could wish for. To be able to do simple calculations like adding two vectors together require multiple calls to the Bohrium API as shown in the Minimal Add Test. There may not be a way

around this at the lowest level, because the calculations are being sent to the GPU, and that require some work. One idea is to create a *dialect* in the World language, that would make it easier for the user. An example of World dialects is seen in the next section, "Calculations with Units".

It would be a benefit from a developer viewpoint to have a strongly reduced subset of all the 1600+ Bohrium functions. It is a huge task to document so many functions, and far the most are not necessary for most applications.

6 Calculations with Units

Within many fields of science, it is a common task to convert between units. This is maybe even more profound in the field of astrophysics, where it is common to be presented with for example solar radius, Astronomical Units, lightyears, and parsecs as units of length, gram and solar masses as units of mass, gigayears as unit of time, electronvolt as unit of energy, etc. etc. Not only is the astronomer required to present the result of a calculation in some exotic unit, but often each calculation involves a mix of many different units for the same physical dimension (of e.g. length, mass, time, etc.). Some areas of astronomy intensively use the cgs (centimetre-gram-second) system of units by tradition, some areas use a mix of what now is easiest to get a grip on, and while the student is maybe most familiar with the SI units, it is easy to get lost.

I was faced with these challenges in my studies, and I wanted to do something about it. It is so easy to make a wrong conversion somewhere in the calculations, and it is often hard to figure out, that there is an error at all, because many results, many numbers are astronomical so to speak.

A good and robust solution uses dialects in World.

6.1 Dialects

Dialects are sub-languages of World. They are like languages within the language, that the programmer can easily define the rules for. It is possible to define ones own sub-language and set the grammar rules, and the language itself utilize this for many of the internal features.

A powerful function to create dialects is the **parse** function, which parses a series according to rules. The series is typical a string or a block of values. To be able to make calculations with units, one would wish to write something like

```
1pc / 1ly
```

to calculate the relation between parsec and lightyears. Maybe one wants to easily find the conversion between $km/\sec$ and kpc/yr (kilo-parsec per year) and so on.

Writing such calculations directly within World gives an error, because values like $1pc$ is not a valid number. But with the help of the KWATZ! datatype, it is possible to load such values into the language, and that can be used to develop a dialect to calculate with units. The lexical analysis within World read in values and recognize their types based on the syntax. 1 is an integer! type, 3.14 is a real! type, etc. When World is asked to **load** a string, a block of values will be created, and each value within the block is given a type. If the syntax isn't recognized (as in the case of e.g. $1pc$), that value is given the type of KWATZ!. It is now possible to parse the block and convert all the values of type KWATZ! to numbers given a conversion table.

I did that one evening, and **gcalc** was born. I mention, that this was done as a little exercise one evening to illustrate the power of dialects, the KWATZ! datatype, and **parse**, which is part of the World language. I know of no other programming language, where I could do this with such little effort.

6.2 gcalc

The **gcalc** solution to the *calculations with units* challenge consists of two small functions and a conversion table. **gcalc** itself is the first function, and the full source is listed in Figure 5.

```
gcalc: func [
    source  [string!]
    /in
        unit
    /local calc mark value rules
][
    rules: [
        any [
            mark:
            set value KWATZ! (
                insert remove mark convert value
            )
            | into rules
            | 1 skip
        ]
        end
    ]
    calc: load source
    parse calc rules
    either unit [
        try [(do calc) / SI/:unit]
    ][
        do calc
    ]
]
```

Figure 5: **gcalc** source.

gcalc takes one argument, **source**, of type string! and return the result of the calculation, default in SI units. It is possible to ask for the result in some other unit with the **/in** refinement and accompanying **unit** argument. The function body does the following.

1. defines the rules,
2. load the source into the **calc** variable,
3. parse the loaded source according to the rules, and
4. finally doing the calculation returning the result.

(If the **/in** refinement was used and therefore the **unit** variable is set, the result is changed into the specified unit before being returned.)

Within the rules, if a value of type KWATZ! is found, that value is changed into a new value returned by the **convert** function, which is the second little function part of the **gcalc** solution. The full source of the **convert** function is listed in Figure 6.

```
convert: func [
    value
    /local nonchar unit
][
    nonchar: make bitset! "0123456789."
    value: to string! value
    unit: pick load copy next' find'/last value nonchar 1
    clear value
    try [SI/:unit * pick load head' value 1]
]
```

Figure 6: **convert** source.

The **convert** function takes one argument, **value**, in some exotic unit and return that same value in SI units. To change to SI units, a SI conversion table is needed, which is shown in Figure 7 for just three units, AU, lightyear, and parsec.

```
SI: make map! [
    AU      1.496e11                ; Astronomical Unit
    ly      9.46047145189709e15     ; Lightyear
    pc      3.08567758e16           ; Parsec
]
```

Figure 7: The SI values.

This SI conversion table can be extended as one see fit. An example of use is shown below.

```
w> include gcalc
w> gcalc "1pc / 1ly"
== 3.261653074785438
```

It is possible to make references to user-defined variables, call functions, etc. within the **gcalc** dialect. All values not of type KWATZ! will just be skipped by the conversion, and only KWATZ! values will be converted according to the SI conversion table.

So for example, if one wishes to calculate the characteristic temperature for the 21 cm hyperfine line, that is utilized in many astronomical observations, one could do the following: define the speed of light, c, Planck's constant, h, and Boltzmann's constant, kB, define "cm" in the SI table, and write

```
w> gcalc "c / 21cm * h / kB"
== 0.0685124692207509
```

To get the result in eV (electronvolt), define it in the SI table, and write

```
w> gcalc/in "c / 21cm * h" 'eV
== 5.903947197896174e-06
```

It is a simple and very useful solution to a very common problem. I know of no other programming language, that can be extended with such functionality with so little effort. I am very happy, that I found this solution, as it helps me a lot in my calculations. An extended SI conversion table is shown in Figure 8.

```
SI: make map! [
    ; Length
    am      1e-18                   ; Attometre
    fm      1e-15                   ; Femtometre
    pm      1e-12                   ; Picometre
    AA      1e-10                   ; Ångstrøm
    nm      1e-9                    ; Nanometre
    um      1e-6                    ; Micrometre
    mm      1e-3                    ; Millimetre
    cm      1e-2                    ; Centimetre
    m       1.0                     ; Metre
    km      1e3                     ; Kilometre
    Mm      1e6                     ; Megametre
    Gm      1e9                     ; Gigametre
    AU      1.496e11                ; Astronomical Unit
    ly      9.46047145189709e15     ; Lightyear
    pc      3.08567758e16           ; Parsec
    kpc     3.08567758e19           ; Kilo Parsec
    Mpc     3.08567758e22           ; Mega Parsec
    Gpc     3.08567758e25           ; Giga Parsec
    ; Mass
    u       1.66053892e-27          ; Atomic mass unit
    g       1e-3                    ; Gram
    kg      1.0                     ; Kilogram
    t       1e3                     ; ton
    ; Time
    ps      1e-12                   ; Picosecond
    ns      1e-9                    ; Nanosecond
    us      1e-6                    ; Microsecond
    ms      1e-3                    ; Millisecond
    s       1.0                     ; Second
    yr      31'556'736.0            ; Year in seconds
    kyr     3.1556736e10            ; Kiloyears in seconds
    Myr     3.1556736e13            ; Megayears in seconds
    Gyr     3.1556736e16            ; Gigayears in seconds
    ; Energy
    eV      1.602'176'565e-19       ; Electronvolt
    keV     1.602'176'565e-16       ; Kilo electronvolt
    MeV     1.602'176'565e-13       ; Mega electronvolt
    GeV     1.602'176'565e-10       ; Giga electronvolt
    TeV     1.602'176'565e-7        ; Tera electronvolt
    J       1.0                     ; Joule
    ; Sun
    m_sun   1.989e30                ; Mass of Sun
    M_sun   1.989e30                ; Mass of Sun
    m_Sun   1.989e30                ; Mass of Sun
    M_Sun   1.989e30                ; Mass of Sun
    r_sun   6.9599e8                ; Radius of Sun
    R_sun   6.9599e8                ; Radius of Sun
    r_Sun   6.9599e8                ; Radius of Sun
    R_Sun   6.9599e8                ; Radius of Sun
]
```

Figure 8: Extended SI conversion table.

7 Plotting

In the summer of 2015, I decided, it was time to include the ability to do plotting in World. I had previously done more and more of my own graph plotting in the R programming language by Ross Ihaka and Robert Gentleman, and I had done some experimentation of plotting in the Rebol programming language by Carl Sassenrath using my own plot routines. Beside this, I had some experience with plotting using tools presented at the university in different courses.

As I use TEX (or LATEX) to write scientific reports, like so many scientists do, I needed to produce PDF (Adobe Portable Document Format) output from the plotting, as this format could easily be included in TEX documents. It is also possible to have tools exporting PNG (Portable Network Graphics) images from PDF files, so I was good covered.

I already had a dialect written in Rebol to produce PS (Adobe PostScript) and PDF output, which I had developed years before for other projects. This software is available online at my physics account at the Niels Bohr Institute at the address http://www.fys.ku.dk/~niclasen/postscript/

I ported the PDF part of this Rebol source to World with little effort, as the two languages are very similar in many ways. This produced a <30kB, 800 lines of World code (as a **wpl2pdf** function), which can create a PDF document from a World Page Layout dialect, I designed.

Next step was to write the **plot** function itself, which should take the user input and call the **wpl2pdf** function, that would produce the PDF output. The **plot** function is 1500 lines of World source, or $\sim$ 40kB, so not huge and bloated as often seen in other languages. Even with its limited size, the **plot** function in World is quite capable, as I will show examples of.

The (in the terminal) accessible help for **plot** is shown in Figure 9.

```
w> ? plot
Usage:
    plot spec /save-wpl save-file /size plot-size /style
        style-word /skip-nan /title plot-title /font overrule
        -font

Description:
    Scientific plotting
    plot is a function!

Arguments:
    spec -- [block!]

Refinements:
    /save-wpl
        save-file -- [file!]
    /size
        plot-size -- [any-type!]
    /style
        style-word -- [word! block!]
    /skip-nan
    /title
        plot-title -- [string! block!]
    /font
        overrule-font -- [word! string!]
```

Figure 9: **plot** help.

plot takes one argument, the **spec** block, in which the user specify all the required information to produce a plot. **plot** also allow several options (the refinement parameters) to overrule some of the defaults.

7.1 Example from "Spot Life"

Giving a data file, `spot-life.dat`, with the following content:

```
Planet   Mr2T4    M1-Ab
1        0.311    0.052
2        0.490    0.082
3        4.182    0.694
4        0.484    0.080
```

, writing the following in World:

```
include plot

plot [
    %spot-life.dat
]
```

, and the plot seen in Figure 10 will be produced.

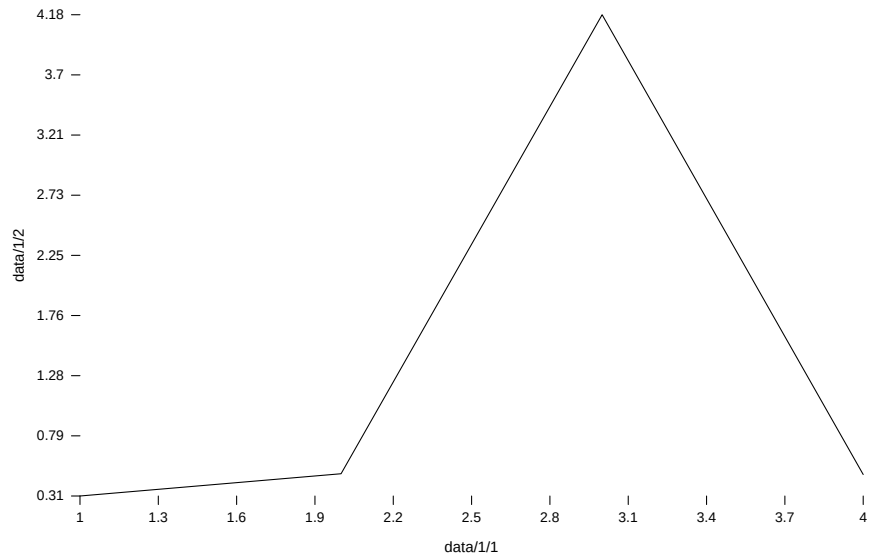

Figure 10: A simple plot.

If **plot** is not given any other parameters than the data file, it will first check, if there is a header line (which there is in this example), skip this and produce a plot with the 1st column along the x-axis, and the 2nd column along the y-axis. The axes will be scaled to the minimum and maximum values.

To produce a nicer plot, writing the following in World will produce the plot seen in Figure 11.

```
plot [
    5x5
    point
    point-width 7
    %spot-life.dat
    Planet  Mr2T4

    x-limit 0 5
    x-ticks 1 4 1
    y-limit 0 4.5
    y-ticks 1 4 1

    x-label "Planet"
    y-label ["M r" ^ "2" " T" _ "eff" ^ "4"]
]
```

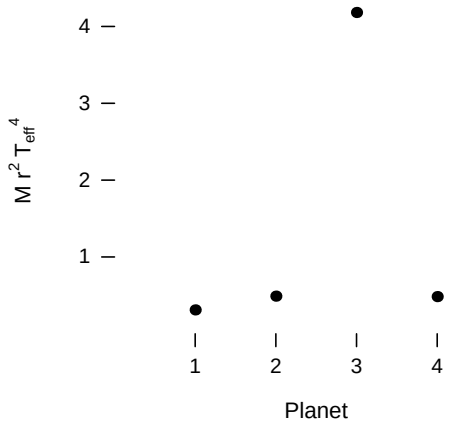

Figure 11: A nicer plot.

The **spec** argument block to **plot** contains a dialect with keywords and values. Some keywords are optional in the cases, where it is given, what is meant by a certain value. Like the first value, 5x5, in the example above, which sets the plot size in inches, so 5 inch by 5 inch here. The final size of the plot in the TEX document can be adjusted, of course, but the plot size influences the size of text relative to graphics. The next line holds the keyword, **point**, which tells **plot**, that points should be drawn for each data point instead of lines between data points.

Yet a more advanced and more readable example is shown below with the resulting plot in Figure 12.

```
plot/font [
    grid
    size 5x5
    point
    point-width 7
    data %spot-life.dat
    x Planet    y Mr2T4

    x-limit 0 5
    x-ticks 1 4 1
    y-limit 0 4.5
    y-ticks 1 4 1

    x-label "Planet"
    y-label ["M r" ^ "2" " T" _ "eff" ^ "4"]

    text 1 0.6 "Mercury"
    text 2 0.8 "Venus"
    text 3 3.8 "Earth"
    text 4 0.8 "Mars"
] "Times"
```

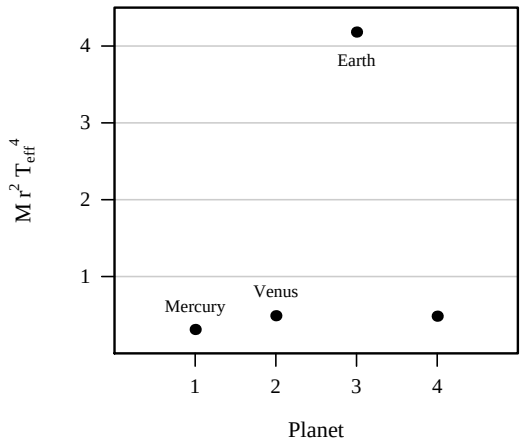

Figure 12: The Spot Life plot.

This example is taken from my paper with the title "Spot Life from Planet's Effective Temperature", which is included in Appendix D. That paper hasn't been presented before this thesis, and I include it here as an example of scientific work, where the World **plot** function can be used.

7.2 Examples from "Periodicity of Sea Ice Extent"

Figure 13 shows yet some features of plotting in World. Colours and transparency is possible.

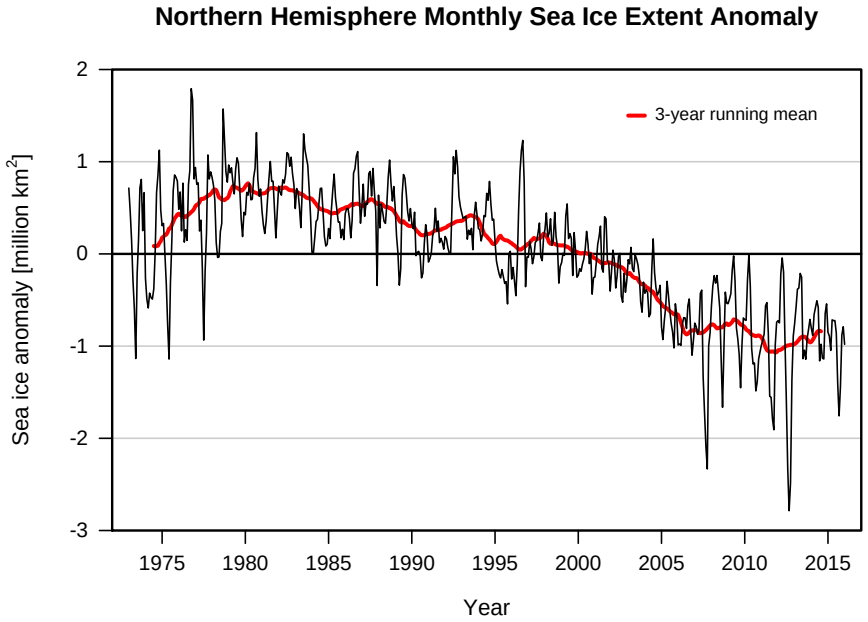

Figure 13: Northern hemisphere sea ice extent anomaly.

The World source to produce the graph in Figure 13 is listed below.

```
plot [
    8x6
    grid
    line-width 1.0
    data %arctic-extent.dat
    pdf %arctic-anomaly.pdf
    year anomaly

    x-limit 1972 2017
    x-ticks 1975 2015 5
    y-limit -3 2
    y-ticks -3 2 1

    line 1.5 [1972 0 2017 0]

    line red 2.5 running-mean
    line red 2.5 [2003 1.5 2004 1.5]
    text 2009.5 1.5 "3-year running mean"

    title "Northern Hemisphere Monthly Sea Ice Extent Anomaly
        "
    text 1994.5 2.25 "Anomaly from 1973-2015 mean"
    x-label "Year"
    y-label ["Sea ice anomaly [million km" ^ "2" "]"]
]
```

The plotting data for the running mean is hold in the **running-mean** variable, which is calculated before the plot. It is possible to combine all such kinds of plotting elements into the final plot.

One more example is shown in Figure 14.

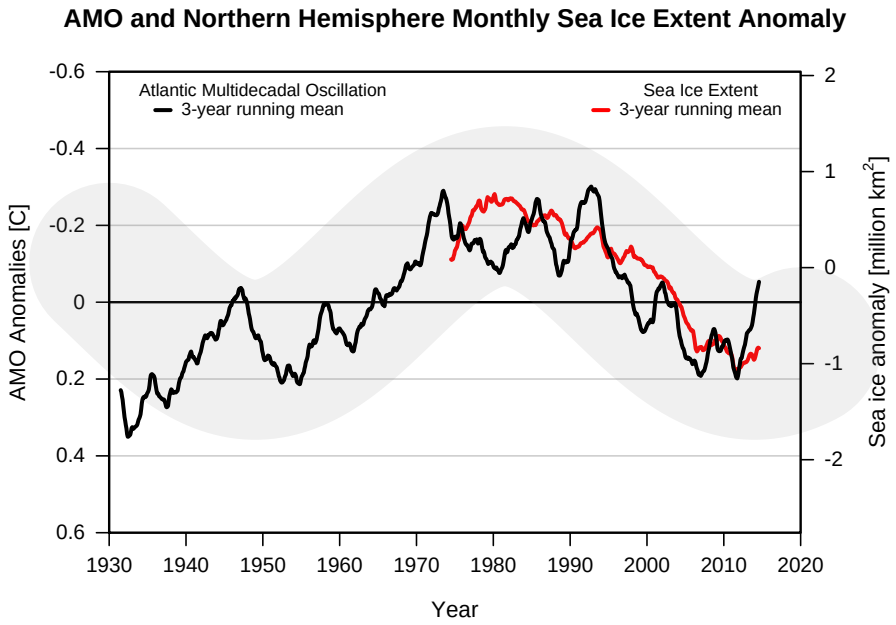

Figure 14: Comparing the Atlantic Multidecadal Oscillation (AMO) with northern hemisphere sea ice extent anomaly.

The World source to produce the graph in Figure 14 is listed below.

```
plot [
    8x6
    grid
    line-width 2.5
    data %../AMO/run-mean-3.dat
    pdf %arctic-anomaly-amo.pdf
    year run-mean-3

    x-limit 1930 2020
    x-ticks 1930 2020 10
    y-limit 0.6 -0.6
    y-ticks 0.6 -0.6 -0.2

    right [
        y-limit -2.76 2.04
        y-ticks -2 2 1
        y-label ["Sea ice anomaly [million km" ^ "2" "]"]
    ]

    line 1.5 [1930 0 2020 0]

    line red 2.5 running-mean
    line black 2.5 [1936 -0.5 1938 -0.5]
    text 1950 -0.55 "Atlantic Multidecadal Oscillation"
    text 1950 -0.5 "3-year running mean"
    line red 2.5 [1993 -0.5 1995 -0.5]
```

```
    text 2007 -0.55 "Sea Ice Extent"
    text 2007 -0.5 "3-year running mean"

    line 100 128.128.128.30 curve

    title "AMO and Northern Hemisphere Monthly Sea Ice Extent
        Anomaly"
    text 1994.5 2.15 "Anomaly from 1973-2015 mean"
    x-label "Year"
    y-label "AMO Anomalies [C]"
]
```

The two examples in this subsection is from a work-in-progress paper with the title "Periodicity of Sea Ice Extent", which can be found in Appendix E. In that work-in-progress can be seen several examples of using the **plot** function in World - twelve plots in all.

7.3 Fitting a Line

Figure 15 shows an example of a double-logarithmic plot of some random data and then fitting a least-square line to the data points. The example also include the use of a plotting style, in this case named "navy-blue", which is defined in the **plot** function.

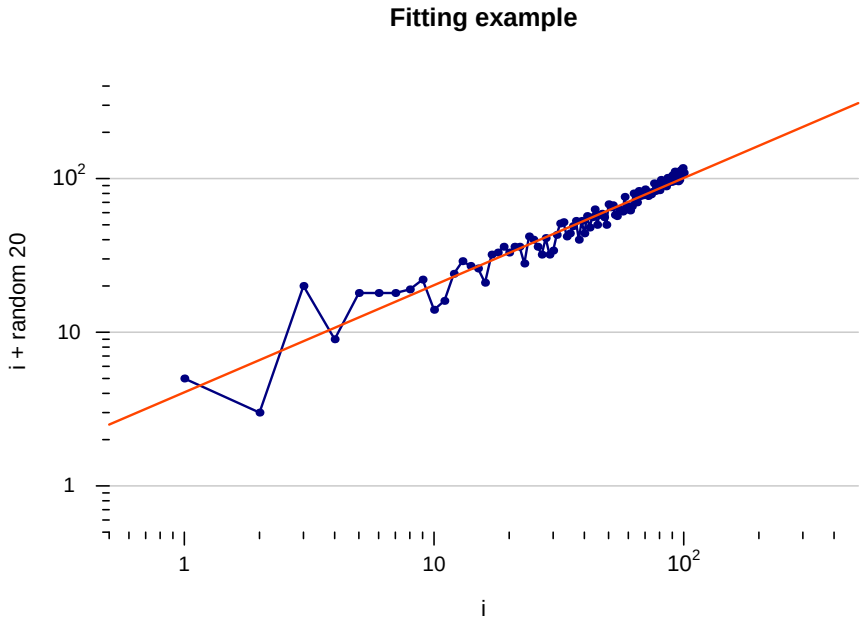

Figure 15: An example of fitting a line to (in this case) random data and presenting the plot with logarithmic axes.

The full World source of the example script to produce the plot in Figure 15 is listed below.

```
World []


include plot


dat: copy []
repeat i 100 [
    append/only dat reduce [i i + random 20]
]


plot [
    style navy-blue
    size 8x6
    point-line
    data dat

    log-xy

    x-limit 0.5 5e2
    y-limit 0.5 5e2

    title "Fitting example"
    x-label "i"
    y-label "i + random 20"

    fit
]
```

When including the **fit** keyword in the **plot** dialect like this, a **fit/least-squares** function will be called by **plot**, and a line will be drawn in the plot. Appendix F lists the full World source of the `fit.w` script including the **fit** context and the **least-squares** function. It is an example of how to write code, that can easily be included in World scripts using the **include** function.

Table 3 lists all the keywords recognized by the World **plot** dialect at time of writing.

Table 3: Plot keywords.

Keyword	**Description**
A4	Set plot size to A4 format
A5	Set plot size to A5 format
backdrop	Set backdrop colour
color	Set colour for lines, crosses, and points
cross	Specify crosses as data points
cross-line	Data points as crosses with connecting lines
data	Specify data source
errorbar	Include error bars
fit	Fit a line
fit-color	Colour of fitting line
fit-limit	Limit fit to range of x values
fit-width	Specify line-width for fitting line
grafik	Include graphics using the grafik dialect
grid	Include grid in plot
grid-color	Specify grid colour
Letter	Set plot size to US letter format
line	Include extra line(s) in plot
line-width	Specify line-width for plot
log-x	Set logarithmic x-axis
log-xy	Set logarithmic x- and y-axis
log-y	Set logarithmic y-axis
pdf	Specify filename for PDF output
point	Specify points as data points
point-line	Data points as points with connecting lines
point-width	Specify width of points in plot
polygon	Include extra polygon(s) in plot
right	Specify axis in right-side of plot
size	Specify plot size
style	Specify a plotting style
text	Include text in plot
text-color	Specify colour of texts
title	Set plot title
type	Set type of data points (cross, point, line, etc.)
x	Specify data values to be used for the x-axis
x-label	Set label for x-axis
x-limit	Limit x data to range of values
x-ticks	Specify ticks along the x-axis
y	Specify data values to be used for the y-axis
y-label	Set label for y-axis
y-limit	Limit y data to range of values
y-ticks	Specify ticks along the y-axis

7.4 Plot with Graphics

The last plotting example in Figure 16 is from a paper with the title "On the Dead Mass Constant" found in Appendix G. It shows how plotting can be combined with other graphics (in this case an ellipse). Creating graphics with the *grafik* dialect is discussed in the next section.

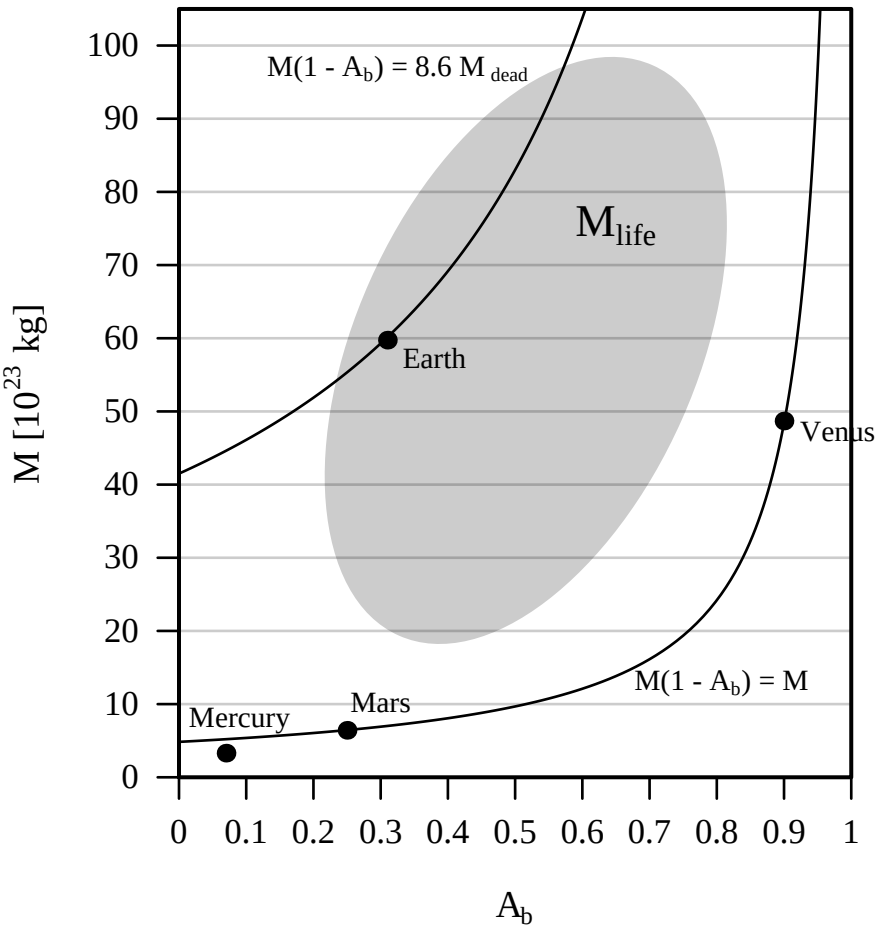

Figure 16: Inner Planets.

8 grafik

In scientific writing, it being for publication or for educational purposes, internal documentation, etc., it is necessary to be able to create graphical illustrations. It could for example be an illustration of a scientific experiment or graphics overlain a graph plot of results or other data.

As PDF output is already possible without much effort in World, as shown in the previous section about plotting, this can also be utilized to create graphical illustrations.

I have developed a basic World dialect named *grafik*, which can be used to create graphical illustrations to be included in TEX or other documents. The dialect is quite new and not fully developed, but it is mentioned here as a proof of concept.

8.1 Basic Graphical Elements

Figure 17 shows some of the basic graphical elements, which the *grafik* dialect is capable of at time of writing.

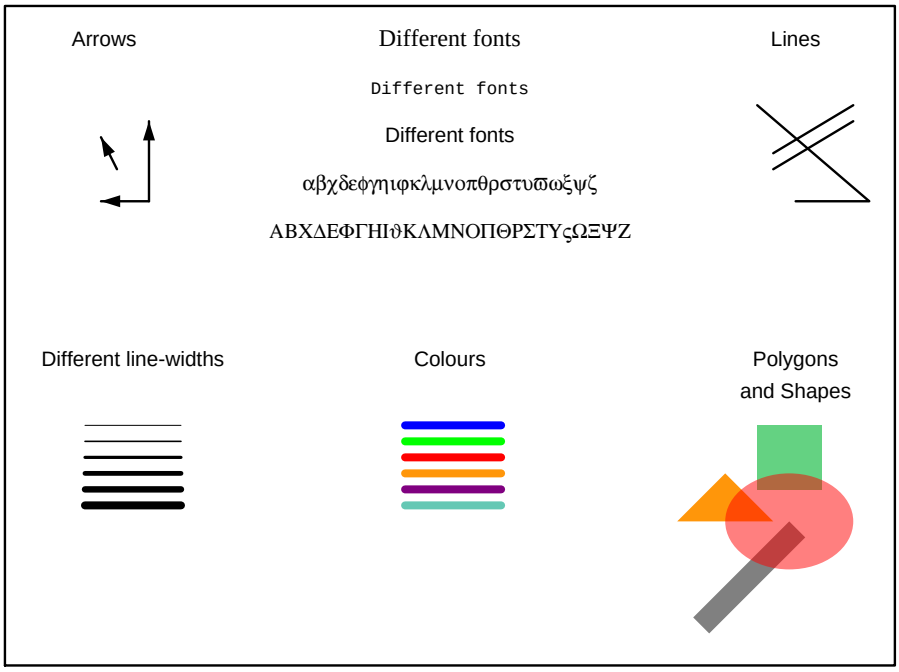

Figure 17: Example output from the World graphical dialect, grafik.

The powerful concept of dialects using the **parse** function in World makes such extensions to the language possible. It can be imagined, how this extension can be built upon to include new geometric shapes and reuse of more complicated figures as known from the Logo language.

The World source code using *grafik* to produce the graphics in Figure 17 is listed below.

```
include grafik/pdf

grafik [
    size 8x6
    line-width 1.5

    ; Frame
    line 10x10 566x10 566x422 10x422 10x10

    text 72x400 "Arrows"
    arrow 100x300 100x350
    arrow 100x300 70x300
    arrow 80x320 70x340

    font [Times 15] text 288x400 "Different fonts"
    font [Courier 11] text 288x370 "Different fonts"
    font [Helvetica 13] text 288x340 "Different fonts"
    font [Symbol 13]
    text 288x310 "abcdefghijklmnopqrstuvwxyz"
    font [Symbol 13]
    text 288x280 "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    font [Helvetica 13] text 504x400 "Lines"
    line 504x300 550x300 480x360
    line 540x360 490x330
    line 540x350 490x320

    text 90x200 "Different line-widths"
    line-width 0.5 line 60x160 120x160
    line-width 1 line 60x150 120x150
    line-width 2 line 60x140 120x140
    line-width 3 line 60x130 120x130
    line-width 4 line 60x120 120x120
    line-width 5 line 60x110 120x110

    text 288x200 "Colours"
    pen blue line 260x160 320x160
    pen green line 260x150 320x150
    pen red line 260x140 320x140
    pen orange line 260x130 320x130
    pen purple line 260x120 320x120
    pen 100.200.180 line 260x110 320x110

    text 504x200 "Polygons"
    pen 100.210.130
    polygon 480x160 520x160 520x120 480x120
    pen orange
    polygon 460x130 490x100 430x100
    pen gray
    polygon 500x100 510x90 450x30 440x40
    pen 255.0.0.128
    ellipsefill 500x100 40x30

    text 200x40 "This is only the beginning!"
]
```

The *grafik* dialect itself is a few hundred lines of code.

8.2 Computational Graphics

The *grafik* dialect supports the ability to compute graphical elements outside the dialect and include it using code within a parenthesis. An example of this is listed in the code below.

```
; Creating a block named knob to hold grafik dialect elements
knob: copy []
; Loop creates eight times two small lines in a semi-circle
repeat n 8 [
    append knob reduce [
        'line
            72 + (30 * cos deg n * 30 - 50)
            72 + (30 * sin deg n * 30 - 50)
            72 + (33 * cos deg n * 30 - 50)
            72 + (33 * sin deg n * 30 - 50)
        'line
            72 + (30 * cos deg n * 30 - 40)
            72 + (30 * sin deg n * 30 - 40)
            72 + (33 * cos deg n * 30 - 40)
            72 + (33 * sin deg n * 30 - 40)
    ]
]
append knob [line-width 4]
repeat n 9 [        ; This loop creates nine dots
    append knob reduce [
        'line
            72 + (31.5 * cos deg n * 30 - 60)
            72 + (31.5 * sin deg n * 30 - 60)
            72 + (31.5 * cos deg n * 30 - 60)
            72 + (31.5 * sin deg n * 30 - 60)
    ]
]


grafik [
    size 2x2
    line 72x66 72x78
    line 66x72 78x72
    text 44x44 "min"
    text 100x44 "max"
    ; The block of code is included in the dialect here
    (knob)
]
```

The result of this code is shown in Figure 18. This could for example be part of an illustration for a scientific instrument with a dial button.

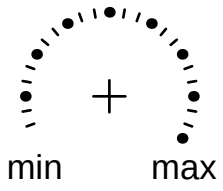

Figure 18: Example output from the World graphical dialect, grafik, showing the possibility to produce dialect source code outside the dialect block and include the code using a parenthesis.

Figure 19 and 20 show examples of the *grafik* dialect being used to illustrate an experimental setup.

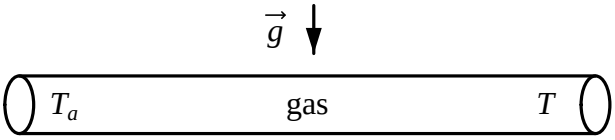

Figure 19: A horizontal tube of gas.

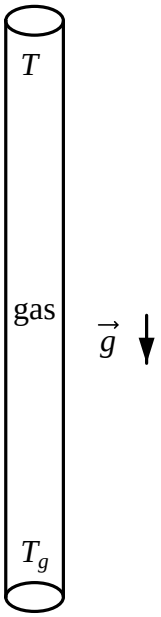

Figure 20: A vertical tube of gas.

In these examples, graphical elements, lines and ellipses, are combined into figures. The examples are from a work-in-progress paper with the title "Lapse Rate Experiment" found in Appendix H.

Table 4 lists all the keywords recognized by the **grafik** dialect at time of writing.

Table 4: Grafik keywords.

Keyword	**Description**
arrow	Draw an arrow
ellipse	Draw an ellipse
ellipsefill	Draw a filled ellipse
font	Specify font for text
landscape	Rotate drawing 90 degrees
line	Draw a line between two or more points
line-width	Specify width of line
pen	Specify colour
polygon	Draw a polygon between three or more points
rotate	Rotate further drawing a number of degrees
size	Specify size of drawing (A4, A5, Letter, inches)
text	Draw text
translate	Translate (move) further drawing
x-limit	Specify clipping in x direction
y-limit	Specify clipping in y direction

9 Multitasking

In some programming situations, we are faced with the requirement to do several tasks at the same time simultaneously. If we program our computer to just solve one task at a time before starting the next, we will not have as responsive systems. In some critical tasks, it is a no-go to finish one task before starting the next. The astronomer (or any other scientist) might be in a situation, where data needs to be read across a communication line (over the internet for example), and then be processed and stored, while the next bunch of data is available to be received. If the sender produce the raw data faster than we can receive and process it, the system will become non-responsive and might lead to buffer overflow and probably crashes.

There are solutions on several layers to solve such situations with the World programming language.

9.1 Tasks

The World programming language has built-in *pre-emptive multitasking* including a *task scheduler*, which will manage task switching and give each task a certain amount of CPU cycles.

It is as easy to define and launch a task as it is to define a function and call it. To illustrate this, consider two tasks, **ping** and **pong**, which will take no arguments and each print a string ten times. The tasks could be defined as seen below.

```
ping: task [] [
    loop 10 [
        prin "ping "
    ]
]

pong: task [] [
    loop 10 [
        prin "pong "
    ]
]
```

The **prin** function used to print the strings will not output a newline, as **print** does. The number of virtual machine instructions, each task is allowed to execute, can be set with the **tasks** function using the **/tick** refinement. So let us set the number of instructions to 50 and launch both task and see, what is output. The result of doing this at the World prompt is shown below.

```
w> tasks/tick 50
== 50
w> ping pong
ping ping ping pong pong pong ping ping ping pong pong pong
     ping ping ping ping pong pong pong pong
```

It is seen, that each task will run long enough to print their strings 3-4 times. Changing the argument to **tasks/tick** to something else than 50 will change the number of instructions (and therefore the time) each task is allowed to run,

before being interrupted by the *task scheduler*. This way of switching between tasks is called *pre-emptive multitasking*.

To make one task hold its execution and let other tasks run, one should call the **wait** function with the argument zero. Now the tasks are defined as shown below.

```
ping: task [] [
    loop 10 [
        prin "ping "
        wait 0
    ]
]

pong: task [] [
    loop 10 [
        prin "pong "
        wait 0
    ]
]
```

The result of launching these tasks at the World prompt is shown below.

```
w> ping pong
ping pong ping pong ping pong ping pong ping pong ping pong
    ping pong ping pong ping pong ping pong
```

It is seen, that each task will print once, and then let other tasks run.

Using tasks is an easy way to create programs, where for example computations will run simultaneously on the same CPU core. We could also imagine one task reading data from a fileserver, while other tasks do further work on the data being read. It is then necessary to have the tasks communicate with each other, and that is done with messages.

9.2 Messages

In the following example, two tasks are defined. **task1** takes one argument, **sec**, and wait for so long, before sending a message to **task2** and quits. **task2** goes into a **while**-loop, until it receives a message. It then prints the message and is done. The World source code is shown below.

```
task1: task [sec] [
    wait sec         ; wait sec amount of seconds
    send id2 "done" ; send a message to id2
]

task2: task [/local msg] [
    while [not msg: receive] [
        print "waiting..."
        wait 0.2     ; wait 0.2 sec.
    ]
    print ["I received:" msg]
]

task1 1      ; launch task1 giving 1 as argument
id2: task2   ; launch task2 and save task-id in id2
```

Running this in the World prompt will produce the output from **task2** seen below.

```
waiting...
waiting...
waiting...
waiting...
waiting...
waiting...
I received: done
```

As is shown, it takes very little code to define and launch tasks, and to send messages between tasks. In some applications, it is a benefit to spread the task load across several CPU cores, either in the same physical computer, or across several computers linked together. To do this, we needs processes, and this will be discussed next.

9.3 Processes

An easy way to spread the task load across several CPU cores is to launch the World programming language several times. The foot-print of World is very small; the source is just around 1MB. Each entity of World can have several tasks, as shown examples of above. Tasks have very small memory foot-print, so many tasks can be launched without running out of memory. Then there is the issue of communicating between World entities (or processes) - what is called interprocess communication.

9.4 Interprocess Communication

Communication between processes can be done using the TCP/IP protocol. The benefit of using TCP/IP is, that each process can be on the same physical computer, or they can be at each end of the physical world, and the internet will then be used to do the transfer of information. In other words, there are no changes to the program, if communicating internal within the same computer or between computers.

In the client/server example shown here, one process acts as a server, which will act on two types of messages.

1. time - the server will return the current time
2. quit - the server will close its communication port and quit

When the server task is started, it will create a listening port and wait for clients to connect to it. In this example, the server will listen on the IP-port 8080.

The World source code for the server task is shown below.

```
server: task [/local cmd p][
    lp: open tcp://:8080
    while [p: read lp] [
        l: to integer! first read/part p 1
        cmd: as string! read/part p l
        print ["server:" l cmd]
        switch cmd [
            "quit" [
                close p
                break
            ]
            "time" [
                a: form now/time
                write p to char! length? a
                write p a
            ]
        ]
        wait 1
        close p
    ]
    close lp
]
```

A client task will then open a communication port to the IP-address, the server is located at, and send commands. If the server is located at the same computer as the client, the IP-address, 127.0.0.1, meaning local-host can be used. The World source code for a client task is shown below.

```
client: task [][
    p: open tcp://127.0.0.1:8080
    write p to char! 4
    write p "time"
    l: to integer! first read/part p 1
    answer: as string! read/part p l
    print ["client:" l answer]
    wait 1
    close p
]
```

In this example, both the server and the client task prints to the terminal. An example of use is shown below.

```
server              ; This will launch the server task
client              ; This will launch one client task
server: 4 time      ; The output from the server
client: 8 13:00:08  ; The output from the client
```

We can imagine several client tasks spread across several computers sending the server task the "time" command and each getting responses in return. Because the TCP/IP protocol is used to communicate between processes, it is also possible to communicate with software written in any other language, that must have support for TCP/IP communication. The server can be closed by sending the "quit" command.

It is my hope, the reader with these examples can see, what little effort it requires in World to have tasks and processes communicate between them, be it on the same computer or across multiple physical computers.

10 Expanding World

We have seen examples of extensions to World in the above with plotting and graphics among other things. It is seen, that it is possible to get far with little effort. The reason for this is the powerful combination of dialects, rich collection of core functions, a large amount of datatypes built in, and minimal syntax among other strong features of the language.

But how well is the language suited to for example implementing a completely new datatype? At one point, I had the need to do calculations with matrices. Matrices is planned for a future release, but they are not implemented yet. I really needed this functionality, so what should I do?

10.1 Matrices

I choose a solution, where I implemented matrices using another datatype, and then programmed a set of functions to be used for calculations with matrices. I include it here to show as a proof of concept, that such a need isn't a large problem and a show-stopper in World.

One very commonly used datatype in World is the block! type. A block is a series of values of different types. Some examples are shown in Figure 21.

```
[a block of words]
[1 word "string" [block within block] 1+2i]
[
    name "John Niclasen"
    occupation "student"
    location "Niels Bohr Institute"
]
```

Figure 21: Examples of World blocks.

I then defined a matrix as a block, where the first element was a complex number (of type complex!) with the real and imaginary parts telling the number of rows and columns in the matrix. The rest of each matrix was filled with rows times columns number of reals (of type real!).

I then programmed the functions listen in Table 5, which was just the ones needed for my calculations.

Table 5: Matrix functions.

Name	**Specification**	**# arguments**
mmul	Matrix multiply.	2
mT	Matrix Transpose.	1
mdet	Matrix Determinant.	1
minv	Inverse matrix, A^(-1).	1
mdiag	Make diagonal matrix.	1
mprint	Print matrix.	1

The implementation was less than 200 lines of code. The full World source is found in Appendix I.

11 The Future of World

In this section, I will briefly reflect on some of the directions and ideas, I have for the World Programming Language. To interface with other technologies, other languages, there are basic two ways: calling from World to the outside, and calling World from the outside.

11.1 Calling from World

Dynamic Linked Libraries

It is fairly easy to define routines in World, as has been shown examples of, to be able to connect with and call functions in libraries programmed in any language, that support dynamic linked libraries. A lot of software packages support this way of operation, so it is obvious to utilize this using World. Interfaces for databases, libcurl (which is a library with a very rich support for all kinds of network protocols), and many other useful things could be developed and shared between users of the language.

Web Services

It is also possible to call web services over the TCP/IP protocol, which is available out-of-the-box with World.

11.2 Calling World

Beside calling World over TCP/IP, we could also imagine World as a dynamic linked library itself. I have already done some testing with this approach, and it is a way to include World as a powerful tool with little foot-print into existing software packages.

11.3 Compiled Dialect

World runs on a virtual machine, which has been developed from scratch for the language. Source code is being compiled into machine code for this virtual machine on-the-fly. To have speed like seen in compiled languages (like C, Fortran, etc.), we could imagine a dialect in World, which would be compiled to native machine code. I have done some research in this area, but there is not a decision made yet. The solution needs to be able to work on different CPUs, as World itself is written in ANSI C with portability in mind.

11.4 NicomDoc, NicomDB, etc.

I have several software solutions, that I initially developed using Rebol. NicomDoc is one, which is a document format to easily create output as HTML and TEX. This thesis is written using NicomDoc, which can be found at http://www.fys.ku.dk/~niclasen/nicomdoc/

It may make sense at some point to port NicomDoc to World, and it will not be a huge task, as World is closely related to Rebol.

In 2004, I finished a thesis about relational databases at Niels Brock. I had developed a database in Rebol, and it would make good sense to port that to World too. Maybe it should wait for a compiled dialect to have top performance.

11.5 World/View and Audio

Recently I have been looking into graphics using OpenGL (also GLSL shaders) and audio with the help of SDL (Simple DirectMedia Layer). SDL is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware via OpenGL. I have a desire to use World to visualize scientific data in real-time and in 3D. That would be possible with this library. There is also the requirement to create simple GUIs (Graphical User Interfaces) for applications across platforms (like there is the X Window System).

What the future will bring, we have to wait and see. It depends also on what connections, I can make with others, who would be interested in technologies, like what World can offer.

12 Conclusion

In this thesis I have presented the World Programming Language, a computer programming language developed by me since 2009. World is among other areas specifically geared to science, and doing this thesis I have investigated programming tasks related to science, and how well World would do in completing such tasks. After giving the background in the introduction, I pointed out programming tasks related to science in general, and astronomy more specific, and what current methods are used in these programming situations. I find, that World is very well suited as a programming tool in many of these areas.

Then I presented a small project related to managing FITS files, that I carried out, before I looked into different areas, where World was tested and used as a programming tool.

I looked into how well suited World is to be used together with Bohrium, an API for parallel computing on GPUs and vector CPUs developed by eScience at the Niels Bohr Institute. I showed, how easy it is to define routines in World to call functions in the Bohrium dynamic linked library, and I showed examples of computations in World, that use these routines calling Bohrium.

I showed, how scientific calculations with units can be done in World with little effort. I presented dialect extensions to World used for scientific plotting of graphs and to produce graphics in general using the *grafik* dialect. And I showed examples of how pre-emptive multitasking and task communication using a messaging system is an integrated part of World, and how processes and interprocess communication is possible with World. I gave an example of how World can be expanded with new functionality, in this case related to a new datatype and matrix operations.

At the end I briefly mentioned what directions World is heading at in the future.

In several sections of this thesis, I refer to examples of scientific papers, I have been working on or are work-in-progress, and where I have used World one way or the other. These works have not been published before, and they are found in the appendices: D "Spot Life", E "Sea Ice Extent", G "Dead Mass", and H "Lapse Rate Experiment".

World has come a long way the last year and a half, while I have been working on this thesis. I have developed many useful tools with World along the way, and I have released new versions of World with better networking among other things.

Today I use World every day as my preferred programming language. I use it in courses at Copenhagen University, and I use it for other development outside of the university.

It is my wish and my hope, that others will find World useful, and that this language can help to make scientific (and other) programming easier.

Programming a computer should be a fun and giving thing to do.

13 References

Bohrium, http://bohrium.bitbucket.org

CTAN, Comprehensive TEX Archive Network, http://tug.ctan.org

FITS at NASA/GSFC, http://fits.gsfc.nasa.gov

GeoGebra, http://www.geogebra.org

IRAF, http://iraf.noao.edu

Kelly Johnson, K.I.S.S. principle, https://en.wikipedia.org/wiki/KISS_principle

Kernighan, Brian W., and Ritchie, Dennis M., The C Programming Language, https://en.wikipedia.org/wiki/The_C_Programming_Language

LaTeX, https://www.latex-project.org

Rebol Language by Carl Sassenrath, http://www.rebol.com

World Programming Language by John Niclasen,
homepage, http://www.world-lang.org
download, https://github.com/Geomol/World

Appendices

A World Functions

?	Print information about words and values.
KWATZ?	True for KWATZ values.
abs	Absolute value.
actor	Define a task after the actor model.
add	Add two values.
all	Evaluate and return at the first false or n...
and?	First value ANDed with the second.
any-block?	True for any-block values.
any-context?	True for any-context values.
any-function?	True for any-function values.
any-paren?	True for any-paren values.
any-path?	True for any-path values.
any-string?	True for any-string values.
any-type?	True for any-type values.
any-word?	True for any-word values.
any	Evaluate and return at the first value that...
append	Append a value to the tail of a series.
arccos	Inverse trigonometric cosine in radians.
arcsin	Inverse trigonometric sine in radians.
arctan	Inverse trigonometric tangent in radians.
arg	Complex argument
as-pair	Combine x and y values into a pair.
as-range	Combine x and y values into a range.
as	Coerce a series into another datatype witho...
back'	Skip series to its previous position.
back	The series at its previous position.
binary?	True for binary values.
bind	Bind block or function to a specified conte...
bitset	Define a bitset of characters.
bitset?	True for bitset values.
block?	True for block values.
break	Break out of a loop, like while.
call	Execute a command to run another process.
callback	Define a callback function with given spec ...
callback?	True for callback values.
cat	Concatenate and print files
cd	Change the active directory path.
change	Change a value in a series.
char?	True for char values.
clear	Remove all values from the current index to...
close	Close a port.
co	Compile a function, operator or block to ex...
comment	Ignore the argument value.
comment?	True for comment values.
compile	Compile a function, operator or block to ex...
compiled?	Tell if a function or block is compiled.
complement	One's complement.
complex?	True for complex values.
compose	Evaluate a block of expressions, only evalu...
context	Define a unique, underived context.
context?	True for context values.
copy	Copy a value.
cos	Trigonometric cosine in radians.
cosh	Hyperbolic cosine in radians.
datatype?	True for datatype values.
date?	True for date values.
debase	Convert a string from a different base repr...
deg	Convert degrees to radians.
dehex	Convert URL-style hex encoded (%xx) strings...
detab	Convert tabs in a string to spaces. (tab si...
di	Translate a compiled function, operator or ...
dirize	Return a copy of the path turned into a dir...
disasm	Translate a compiled function, operator or ...
divide	First value divided by the second.
do	Evaluate a block, file, function, or any ot...
does	Define a function that has no arguments.
dt	Delta-time -- Time the evaluation of a bloc...
dump-obj	Return a block of information about a conte...
echo	Write arguments to the standard output
either	If condition is true, evaluate the first bl...

```
email?          True for email values.
empty?          True if a series is empty.
enbase          Convert a string to a different base repres...
equal?          True if the values are equal.
error?          True for error values.
exit            Exit a function, return no value.
exp             Raise Euler's number e to the power.
file?           True for file values.
find'           Skip to found value in a series.
find            Find a value in a series.
first           First value of a series.
for             Evaluate a block over a range of values.
forall          Evaluate a block for every value in a serie...
foreach         Evaluate a block for each value(s) in a ser...
form            Convert a value into a human-readable strin...
free-all        Free a list of World resources.
free            Free a World resource.
func            Define a function with given spec and body.
function?       True for function values.
get-path?       True for get-path values.
get-word?       True for get-word values.
get             Get the value of a word.
glob            Generate pathnames matching a pattern
greater-or-equal?  True if the first value is greater than or
    ...
greater?        True if the first value is greater than the...
grep            Print lines matching a pattern.
halt            Stop evaluation of task.
has             Define a function that has local variables ...
hd              Hexadecimal dump
head'           Skip series to its head.
head            The series at its head.
head?           True if a series is at its head.
help            Print information about words and values.
if              If condition is true, evaluate the block.
image?          True for image values.
immediate?      True for any-word values.
include         [ 'file [file! word! path!] ]
index?          Index number of the current position in the...
input           Inputs a string from the console.
insert          Insert a value into a series.
integer?        True for integer values.
issue?          True for issue values.
join            Concatenate values.
kill            Terminate a task.
l               List directory contents in long format.
last            Last value of a series.
length?         Length of a series from the current positio...
lesser-or-equal?  True if the first value is less than or equ
    ...
lesser?         True if the first value is less than the se...
library?        True for library values.
license         Print the World/Cortex license agreement.
list?           True for list values.
lit-path?       True for lit-path values.
lit-string?     True for lit-string values.
lit-word?       True for lit-word values.
ln              Natural (base e) logarithm.
load-library    Load a dynamic library.
load            Load a file or string. Bind block to global...
log             Base 10 logarithm.
logic?          True for logic value.
loop            Evaluate a block a specified number of time...
lowercase       Convert string of characters to lowercase.
ls              List directory contents.
make            Construct a value of a specified datatype.
map             Applies function to successive sets of argu...
map?            True for map values.
max             The greater of two values.
min             The lesser of two values.
mod             Remainder of first value divided by second.
mold            Convert a value to a World-readable string.
more            Opposite of less.
more?           True if a series isn't at its tail.
```

```
multiply        First value multiplied by the second.
native-op       Define a native operator with given spec an...
native          Define a native function with given spec an...
negate          Change the sign of a number.
newline?        State of the newline marker within a block.
next'           Skip series to its next position.
next            The series at its next position.
node            Define a node
node?           True for node values.
none?           True for none values.
not-equal?      True if the values are not equal.
not             Logic complement.
now             Local date and time.
number?         True for number values.
open-port       Open a port.
open            Open a port.
operator        Define an operator with given spec and body...
operator?       True for operator values.
or?             First value ORed with the second.
pair?           True for pair values.
paren?          True for paren values.
parse           Parse a series according to rules.
path?           True for path values.
percent?        True for percent values.
pick            Value at the specified position in a compon...
poke            Change a value at the given index.
port?           True for port values.
power           First number raised to the second.
prin            Output a value with no newline.
print           Output a value followed by a newline.
probe           Print a molded value and return that same v...
pwd             Return working directory name.
q               Stop evaluation and exit World.
query           Return information about a file.
quit            Stop evaluation and exit World.
random          Random value of the same datatype.
range?          True for range values.
read            Read from a file, url, or console port.
real?           True for real values.
receive         Receive a message.
recycle         Recycle unused memory.
reduce          Evaluate expressions and return multiple re...
refinement?     True for refinement values.
remove          Remove value(s) from a series.
repeat          Evaluate a block a number of times.
replace         Replace the search value with the replace v...
retain-all      Retain a list of World resources.
retain          Retain a World resource.
return          Return a value from a function.
reverse         Reverse a series.
rotate          Bit rotate a value.
round           Round a numeric value. Halves round up (awa...
routine         Define a library routine
routine?        True for routine values.
run             Run a World script at its location.
same?           True if the values are identical.
save            Save to a file
scalar?         True for scalar values.
second          Second value of a series.
select          Find a value in a series and return the val...
send            Send a message and return the same message.
series?         True for series values.
set-newline     Set or clear the newline marker within a bl...
set-path?       True for set-path values.
set-word?       True for set-word values.
set             Set a word or block of words to specified v...
shift           Bit shift a value.
sin             Trigonometric sine in radians.
sinh            Hyperbolic sine in radians.
skip'           Skip series forward or backward from the cu...
skip            Series forward or backward from the current...
sort            Sort a series.
source          Print the source code for a word.
split-path      Splits a file or URL path. Returns a block ...
```

```
sqrt            Square root of a number.
stats           System statistics. Default is to return tot...
strict-equal?   True if the values are equal and of the sam...
string?         True for string values.
struct          Define a structure.
subtract        Second value subtracted from the first.
swap-bytes      Toggle between little-endian and big-endian...
switch          Select a choice and evaluate the block that...
tag?            True for tag values.
tail'           Skip series to the position after the last ...
tail            The series at the position after the last v...
tail?           True if a series is at its tail.
tan             Trigonometric tangent in radians.
tanh            Hyperbolic tangent in radians.
task-id?        True for task-id values.
task            Define a task with given spec and body.
task?           True for task values.
tasks           System tasks. Default is to show all tasks.
test            Run tests.
third           Third value of a series.
time?           True for time values.
to-deg          Convert radians to degrees.
to-local-file   Convert a World file path to the local syst...
to-world-file   Convert a local system file path to a World...
to              Construct a new value after conversion.
trace           Control evaluation tracing.
trim            Remove whitespace from a string. Default re...
try             Try to DO a block.
tuple?          True for tuple values.
type?           Value's datatype.
typeset?        True for typeset values.
unset?          True for unset values.
until           Evaluate a block until it is true.
uppercase       Convert string of characters to uppercase.
url?            True for url values.
value?          True if the word has been set.
vector          Define a vector.
vector?         True for vector values.
wait            Wait for a duration, a certain time, 'messa...
while           While a condition block is true, evaluate a...
word?           True for word values.
write           Write to a file.
xor?            First value XORed with the second.
zero?           True if a number is zero.
```

B World Source for Loading FITS Files

```
World [
    Title:  "FITS file util"
    Author: "John Niclasen"
]


ds9: func [
    'file
][
    if file! <> type? file [file: to file! file]
    if %.fits <> skip tail file -5 [append file %.fits]
    call append copy "ds9 -fits " as string! file
]


read-header: func [
    file    [file!]
    /all
    /local fh line
][
    fh: make port! file
    open fh
    line: as string! copy/part fh 80
    while [not find/case/match line "END"] [
        if line/1 <> #" " or all [print line]
        line: as string! copy/part fh 80
    ]
    if all [print line]
    close fh
]


keywords: make map! [
    SIMPLE      Simple
    BITPIX      BitPix
    NAXIS       nAxis
    NAXIS1      nAxis1
    NAXIS2      nAxis2
    EXTEND      Extend
    EQUINOX     Equinox
    RADECSYS    RaDecSys
    CTYPE1      CType1
    CUNIT1      CUnit1
    CRVAL1      CRVal1
    CRPIX1      CRPix1
    CTYPE2      CType2
    CUNIT2      CUnit2
    CRVAL2      CRVal2
    CRPIX2      CRPix2
    EXPTIME     ExpTime
    GAIN        Gain
    SATURATE    Saturate
    SOFTNAME    SoftName
    SOFTVERS    SoftVers
    SOFTDATE    SoftDate
    SOFTAUTH    SoftAuth
    SOFTINST    SoftInst
    AUTHOR      Author
    ORIGIN      Origin
    DATE        Date
    COMBINET    CombineT
    OBJECT      Object
    RESAMPT1    ResampT1
    CENTERT1    CenterT1
    PSCALET1    PScaleT1
    RESAMPT2    ResampT2
    CENTERT2    CenterT2
    PSCALET2    PScaleT2
    CHECKSUM    CheckSum
```

```
    DATASUM         DataSum
    DATA            _Data
]


load-fits: func [
    "Load a FITS file."
    file    [file!]
    /header "Only load header"
    /local fh line lines blk word keyword value buf result
        data-size x y
    ][
    blk: copy []
    fh: make port! file
    open fh
    while [
        lines: 0
        ;
        ; Read header
        ;
        while [
            lines: lines + 1
            line: as string! copy/part fh 80
            not find/case/match line "END"
        ] [
            if all [
                not find/case/match line "COMMENT "
                not find/case/match line "HISTORY "
                line/1 <> #" "
            ] [
                word: first load copy/part line 8
                keyword: select keywords word
                append blk to set-word! either keyword [
                    keyword] [word]
                skip' line 10
                value: either line/1 = #"'" [
                    next' line
                    trim copy/part line find line #"'"
                ][
                    pick to block! copy/part line 20 1
                ]
                switch word [
                    DATASUM
                    DATE
                    SOFTDATE
                    SOFTVERS    [value: load value]
                ]
                append blk either word! = type? value [
                    to lit-word! value
                ][
                    value
                ]
            ]
        ]
        ;
        ; Skip to data
        ;
        if lines // 36 > 0 [
            copy/part fh 36 - (lines // 36) * 80
        ]
        buf: copy/part fh 80
        find/match as string! buf "XTENSION= 'IMAGE"
    ] []
    ;
    ; Read data
    ;
    append blk [data: none]
    result: make context! blk
    if not header [
        either result/nAxis = 0 [
            parse result/DetWin1 [
                thru ":" copy x to ","
                thru ":" copy y to "]"
                to end
```

```
            ]
            x: first load x
            y: first load y
        ][
            x: result/nAxis1
            y: result/nAxis2
        ]
        any [
            if result/BitPix = 16 [
                data-size: x * y * 2
                append buf copy/part fh data-size - 80
                result/data: as reduce ['vector! 'sint16 data
                    -size / 2] buf
            ]
            if result/BitPix = 32 [
                data-size: x * y * 4
                append buf copy/part fh data-size - 80
                result/data: as reduce ['vector! 'sint32 data
                    -size / 4] buf
            ]
        ]
    ]
    close fh
    result
]
```

C World Source for Bohrium Interface

```
World [
    Title:  "Bohrium"
    Author: "John Niclasen"
]

bohrium: true

libbhc: load-library %/home/john/.local/lib/libbhc.so


;
; Common runtime methods
;
bh-runtime-flush: routine [
    "Execute all pending instructions."
    libbhc "bh_runtime_flush" []
    void
]


; integer
bh-integer-new-range: routine [
    "Construct a new array with sequential numbers."
    [typecheck]
    libbhc "bh_multi_array_uint64_new_range" [
        size    [integer!] uint64
    ]
    pointer handle!
]


bh-integer-new-view: routine [
    "Construct a new array from an existing view."
    [typecheck]
    libbhc "bh_multi_array_uint64_new_view" [
        source  [handle!]   pointer
        rank    [integer!]  uint64
        start   [integer!]  sint64
        shape   [vector!]   pointer
        stride  [vector!]   pointer
    ]
    pointer handle!
]


bh-integer-destroy: routine [
    "Destroy the array and release resources."
    [typecheck]
    libbhc "bh_multi_array_uint64_destroy" [
        self    [handle!] pointer   "Array"
    ]
    void
]


; real
; Issue a sync instruction to ensure data is present in local
    memory space
bh-real-sync: routine [
    "Sync the current view."
    [typecheck]
    libbhc "bh_multi_array_float64_sync" [
        self    [handle!] pointer   "Array"
    ]
    void
]


bh-real-get-data: routine [
    "Gets the data pointer from a view."
```

```
    [typecheck]
    libbhc "bh_multi_array_float64_get_data" [
        self    [handle!] pointer   "Array"
    ]
    pointer handle!
]


bh-real-new-ones: routine [
    "Construct a new one-filled array."
    [typecheck]
    libbhc "bh_multi_array_float64_new_ones" [
        rank    [integer!] uint64   "Number of dimensions"
        shape   [vector!] pointer   "Shape of array"
    ]
    pointer handle!
]


bh-real-new-empty: routine [
    "Construct a new empty array of 64 bit floats."
    [typecheck]
    libbhc "bh_multi_array_float64_new_empty" [
        rank    [integer!] uint64   "Number of dimensions"
        shape   [vector!] pointer   "Shape of array"
    ]
    pointer handle!
]


bh-real-destroy: routine [
    "Destroy the array and release resources."
    [typecheck]
    libbhc "bh_multi_array_float64_destroy" [
        self    [handle!] pointer   "Array"
    ]
    void
]


;
; Copy methods
;
bh-real-identity-integer: routine [
    "Make into floats."
    [typecheck]
    libbhc "bh_multi_array_float64_identity_uint64" [
        out     [handle!] pointer
        rhs     [handle!] pointer
    ]
    void
]


;
; Binary methods
;
bh-real-add: routine [
    "Addition."
    [typecheck]
    libbhc "bh_multi_array_float64_add" [
        out [handle!] pointer   "Output"
        lhs [handle!] pointer   "lhs"
        rhs [handle!] pointer   "rhs"
    ]
    void
]
```

D Spot Life

Spot Life from Planet's Effective Temperature

By John Niclasen

April, 2016

Abstract. I suggest a method to determine, if an Earth-like planet in the habitable zone around a star supports life. Through a few measurable parameters for extrasolar planets, I give a simple formula, that will do the trick.

D.1 Introduction

There is a boom of newly discovered planets around other stars in the Milky Way these years. Better space-based and ground-based observatories is the cause of all these new discoveries, and new observational projects are launched, which will lead to even more discoveries. Such a project is run by the Stellar Observations Network Group (SONG), which through the microlensing observation method can find Earth-like and even smaller planets around other stars.

It is a very interesting question to determine, if these planets support life or not.

Table 6: Symbols used

Symbol	Description
A_b	Bond albedo
L	Star's luminosity
M	Mass of planet
r	Planet's distance to its star
T_{eff}	Planet's effective temperature
σ	Stefan-Boltzmann constant

D.2 How to Spot

There seem to be a simple equation, that holds for planets without life:

$$M\ (1 - A_b) \approx constant \tag{1}$$

That is, the mass of the planet times the amount of absorption of solar radiation is a constant. The amount of absorption is 1 minus the Bond albedo for the planet.

If the Bond albedo is not known, it can be found from the effective temperature, which is given as:

$$T_{eff} = \left(\frac{L}{4\pi\ r^2}\ \frac{(1 - A_b)}{4\sigma} \right)^{1/4}$$

$$\Leftrightarrow \quad (1 - A_b) = \frac{16\pi\sigma\ r^2}{L}\ T_{eff}^4 \tag{2}$$

Substituting Eq. (2) into Eq. (1) leads to:

$$M\ r^2\ T_{eff}^4 \approx constant \tag{3}$$

, as the luminosity, L, is constant for planets around the same star.

D.3 Discussion

In Figure 1 is plotted the result for the inner four planets. M is in units of Earth mass, r in astronomical units (AU), T_{eff} in $Kelvin$ and I multiplied by 10^{-9} to get values less than 5.

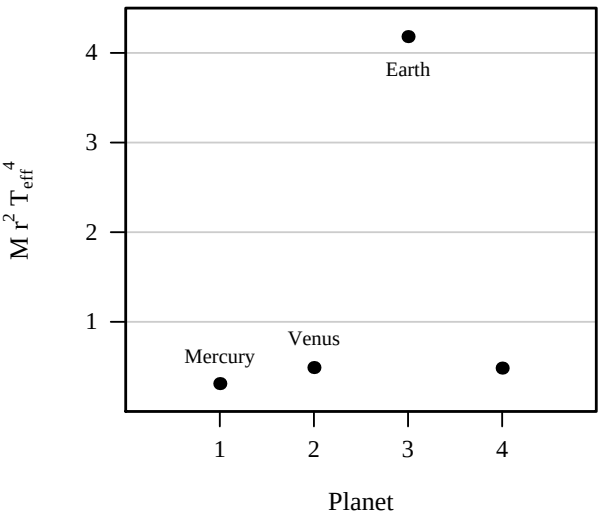

Figure 22: Our Solar System

Even if Mercury isn't in the habitable zone, Eq. (3) still gives a value close to Venus and Mars, a good test of the formula. Eq. (3) gives a very different value for the Earth compared to the other inner planets, and that's because life has changed the atmosphere on Earth, so its effective temperature is much higher, than it would be without life.

There is good reason to assume, the same would be the case for planets with life in other solar systems.

The question now is, if an effective temperature can be determined for extrasolar planets?

D.4 Conclusion

A simple formula was given, that will help determine, if an Earth-like planet in the habitable zone around a star supports life.

We already know the mass and distance to the star to a good precision for many exoplanets. Is it possible also to determine an effective temperature, so the formula can be used?

E Sea Ice Extent

Periodicity of Sea Ice Extent

By John Niclasen

April, 2016

E.1 Early Satellite Data

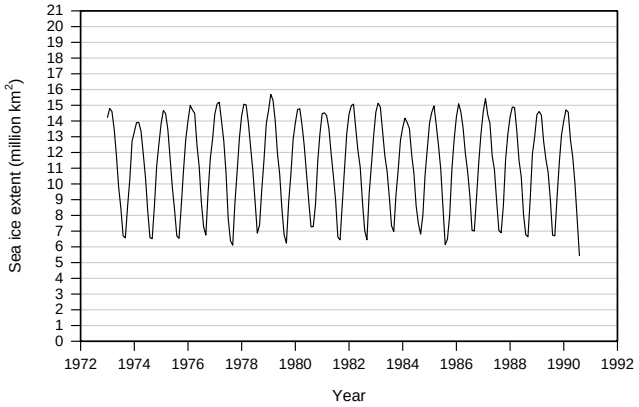

Figure 23: Northern hemisphere sea ice extent, 1973-1990.

Data source: http://nsidc.org/data/g00917 Listed platforms are: aircraft, ground stations, ground-based observations, satellites, ships. Listed sensors are: digitizer, visual observations.

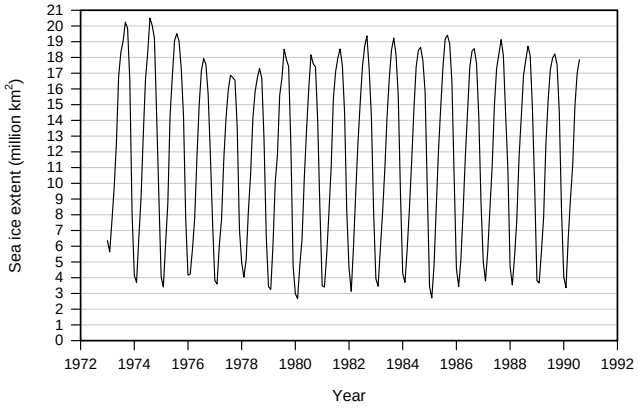

Figure 24: Southern hemisphere sea ice extent, 1973-1990.

Data from same source as in Figure 23.

E.2 Satellite Data 1978-2016

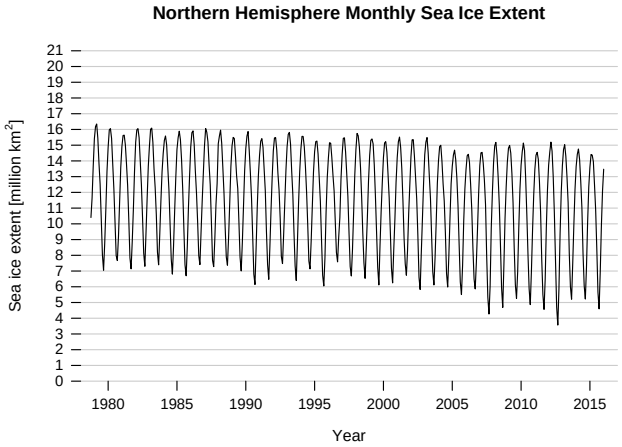

Figure 25: Northern hemisphere sea ice extent, 1978-2016.

Data source: http://nsidc.org/data/g02135. Listed platforms are: DMSP, DMSP 5D-3/F17, NIMBUS-7, satellites. Listed sensors are: SMMR, SSM/I, SSMIS.

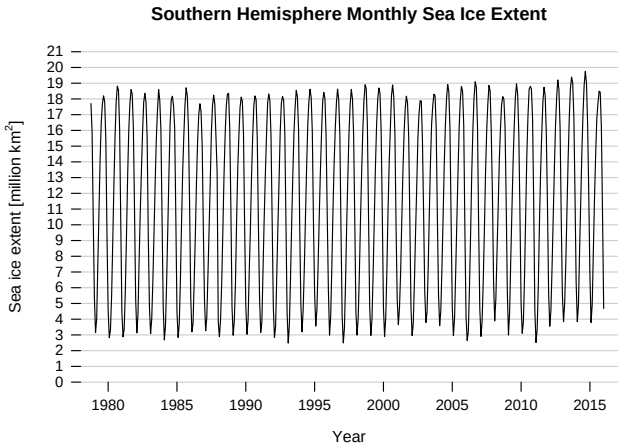

Figure 26: Southern hemisphere sea ice extent, 1978-2016.

Data from same source as in Figure 25.

E.3 A Combined Dataset

If p_i is an extent value in the early satellite data, and q_i is an extent value in the new 1978-2016 satellite data, then the adjusted extent value in the early satellite data, pp_i, is found using Eq. (4).

$$pp_i = Ap_i + B \tag{4}$$

, where A and B is given by Eq. (5) and Eq. (6).

$$A = \frac{\langle q_{\max}\rangle - \langle q_{\min}\rangle}{\langle p_{\max}\rangle - \langle p_{\min}\rangle} \tag{5}$$

$$B = \langle q_{\max}\rangle - A\langle p_{\max}\rangle \tag{6}$$

Northern Hemisphere

The above described method to find the adjustment parameters, A and B, for the early dataset to combine it with the late dataset into a new full dataset for the period 1973-2016, gave these results for the norther hemisphere:

$$A = 1.056941648192076$$
$$B = 0.123270757898819$$

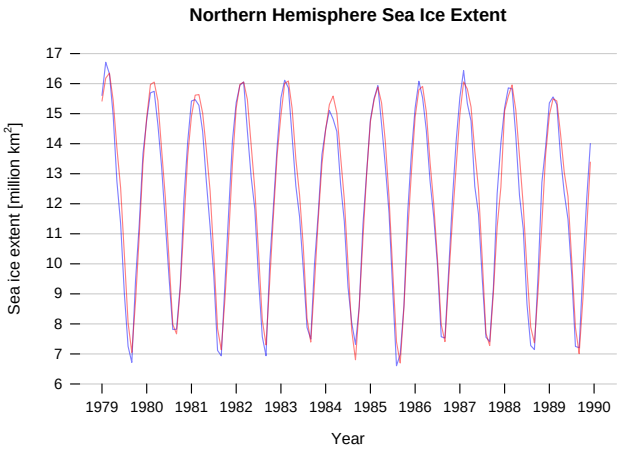

Figure 27: A comparison of the early and late dataset for the northern hemisphere to verify the adjustment of the early one.

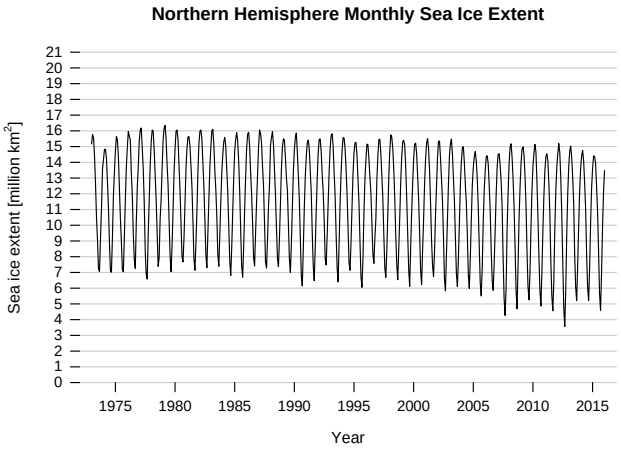

Figure 28: The combined dataset for the northern hemisphere.

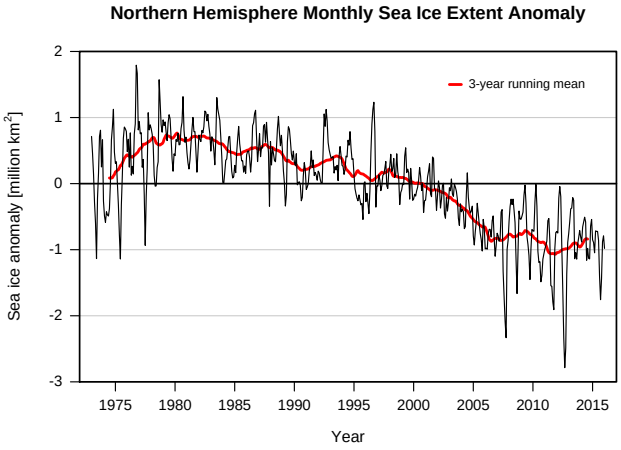

Figure 29: Northern hemisphere sea ice extent anomaly with a 3-year running mean.

The date for each calculated value in the running mean is in the middle of the three years.

Southern Hemisphere

The method to find the adjustment parameters, A and B, for the early dataset to combine it with the late dataset into a new full dataset for the period 1973-2016, gave these results for the souther hemisphere:

$$A = 0.995315462226912$$
$$B = -0.336275617355696$$

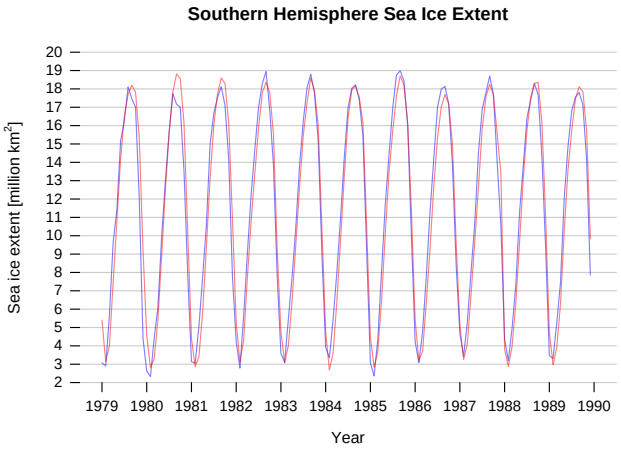

Figure 30: A comparison of the early and late dataset for the southern hemisphere to verify the adjustment of the early one.

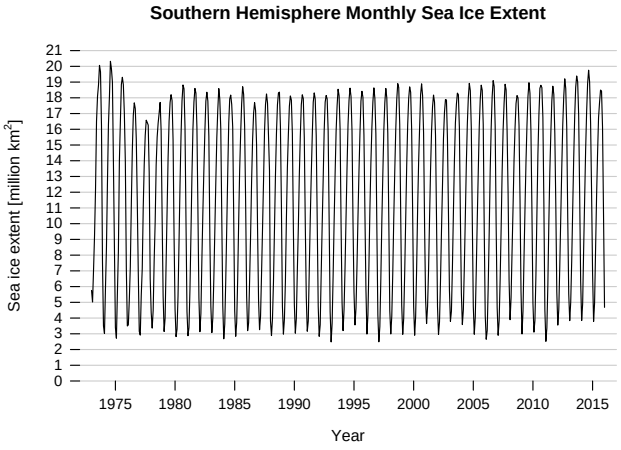

Figure 31: The combined dataset for the southern hemisphere.

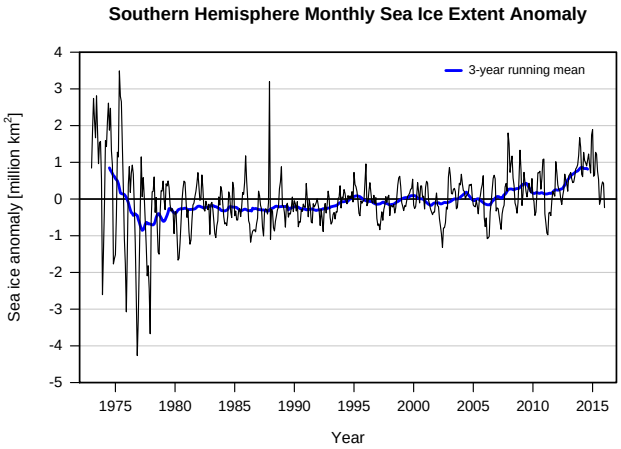

Figure 32: Southern hemisphere sea ice extent anomaly with a 3-year running mean.

The date for each calculated value in the running mean is in the middle of the three years.

E.4 Periodicity

AMO

Atlantic Multidecadal Oscillation (AMO)
North Atlantic Oscillation (NAO)

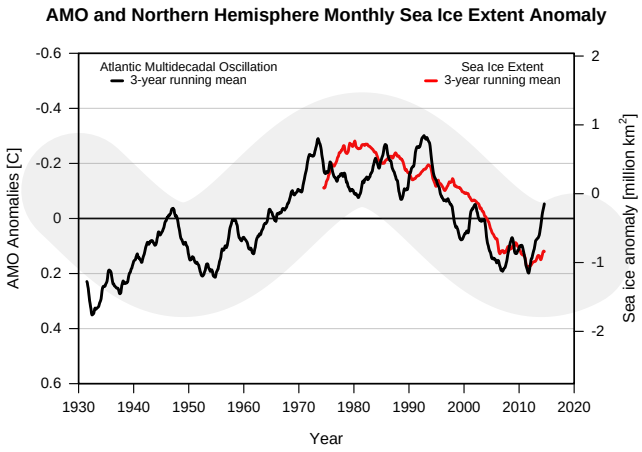

Figure 33: Comparing the Atlantic Multidecadal Oscillation (AMO) with northern hemisphere sea ice extent anomaly. Notice the y-axis for the AMO is increasing cold up. A 65-year period sine curve is overlaid.

Bipolar Seesaw

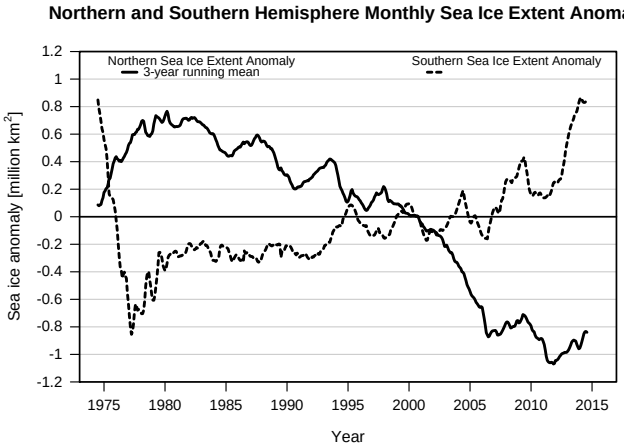

Figure 34: A comparison of the northern and southern hemisphere sea ice extent anomaly.

Only the 3-year running means are shown. The date for each calculated value in the running means is in the middle of the three years.

F World Source for Fit Script

```
World [
    Title:      "Data Fitting"
    Date:       18-Jan-2015
    Version:    1
    File:       %fit.w
    Author:     "John Niclasen"
    Purpose:    {
        Linear Regression or Least-Squares Fit for a line
    }
]


if value? 'fit [exit]


fit: context [


a: 0.0
b: 0.0
r2: 0.0
n: 0
sigma-y: 0.0
sigma-a: 0.0
sigma-b: 0.0


least-squares: func [
    "Linear Regression, y = A + Bx"
    coords  [block!]    "[x1 y1 x2 y2 ...]"
    /quiet
    /local sum-x sum-y sum-x2 sum-xy d mean-y sum-squares
][
    ; See Taylor p. 183 and 197-198
    sum-x2: sum-x: sum-y: sum-xy: 0.0
    foreach [x y] coords [
        sum-x:  x + sum-x
        sum-y:  y + sum-y
        sum-x2: x * x + sum-x2
        sum-xy: x * y + sum-xy
    ]
    n: (length? coords) / 2
    ; Delta = N sum x^2 - (sum x)^2
    d: (n * sum-x2) - (sum-x * sum-x)
    ; A = {sum x^2 sum y - sum x sum xy over Delta}
    a: (sum-x2 * sum-y) - (sum-x * sum-xy) / d
    ; B = { N sum xy - sum x sum y over Delta}
    b: (n * sum-xy) - (sum-x * sum-y) / d
    ; Uncertainties
    ; simga_y = sqrt ({1 over N - 2} sum_(i = 1)^N (y_i - A -
        B x_i)^2)
    mean-y: sum-y / n
    sum-squares: 0.0
    sigma-y: 0.0
    foreach [x y] coords [
        sigma-y: y - a - (b * x) ** 2 + sigma-y
        sum-squares: y - mean-y ** 2 + sum-squares
    ]
    r2: 1 - (sigma-y / sum-squares)
    sigma-y: sqrt sigma-y / (n - 2)
    sigma-a: sigma-y * sqrt sum-x2 / d
    sigma-b: sigma-y * sqrt n / d
    if not quiet [
        print [
            newline
            "A, y0" tab tab a newline
            "B, dy / dx" tab b newline
            "sigma y" tab sigma-y newline
            "sigma A" tab sigma-a newline
```

```
            "sigma B" tab sigma-b newline
            "r2" tab tab to percent! round/to r2 1e-4 newline
        ]
    ]
]


]
```

G Dead Mass

On the Dead Mass Constant

By John Niclasen

April, 2016

Abstract. The *dead mass constant* for Earth-like planets without life in the habitable zone is studied. Universal formulas to calculate the *absorption mass* of all Earth-like planets in the habitable zone are sought, both from the Bond albedo and the effective temperature.

Knowing the *absorption mass* of such planets can help determine, if there is life or not.

G.1 Introduction

Table 7 is a list of the symbols used in this study.

Table 7: Symbols used

Symbol	Description
A_b	Bond albedo
L	Star luminosity
$L_\odot$	Solar luminosity
M	Mass of planet
M_κ	Absorption mass
M_{dead}	Dead mass
M_{life}	Life mass
$M_\oplus$	Mass of Earth
r	Planet's distance to its star
T_{eff}	Planet's effective temperature

In the study *"Spot Life from Planet's Effective Temperature"* (Niclasen, 2013), a formula for a *weighted mass* was given. This mass can be called *absorption mass*, as it is weighted by the amount of radiation absorbed from the star, and it will be denoted by M_κ.

$$M_\kappa = M\ (1 - A_b) \tag{7}$$

It was argued, that this quantity is a constant for planets without life in the habitable zone around a star. The present study will specify the constant both for the Solar System and for Extrasolar Systems, and it will be studied, what this constant tells us.

G.2 Parameters

Table 8 shows some measured and calculated values for the four inner planets in our Solar System.

Table 8: Inner Planets

Parameter	**Mercury**	**Venus**	**Earth**	**Mars**
M $[10^{23}\ kg]$	3.302	48.685	59.736	6.4185
M $[M_\oplus]$	0.0553	0.815	1	0.107
A_b	0.068	0.90	0.306	0.250
$(1 - A_b)$	0.932	0.10	0.694	0.750
M_κ $[10^{23}\ kg]$	3.08	4.87	41.5	4.81
M_κ $[M_\oplus]$	0.0515	0.0815	0.694	0.0803
”	$\sim 5\%$	$\sim 8\%$	$\sim 70\%$	$\sim 8\%$

The last line in Table 8 gives the *absorption mass*, M_κ, in percentage of the Earth mass, $M_\oplus$.

G.3 Discussion

For planets without life in the habitable zone around a star:

$$M_{dead} = M_\kappa \approx constant \tag{8}$$

By looking at the two planets without life in the habitable zone in our Solar System, Venus and Mars, a value for M_{dead} can be estimated:

$$M_{dead} \approx 4.84 \pm 0.03 \times 10^{23}\ kg \tag{9}$$

The value of M_κ for the Earth, which has life, is ~ 8.6 times larger than M_{dead}. The atmospheres of Venus and Mars is almost purely carbon dioxide ($\sim 96.5\%$ for Venus and $\sim 95.32\%$ for Mars by latest measures). On the other hand, Earth, which has had life for billions of years, now has less than 0.04% carbon dioxide left in its atmosphere. Under assumption, that the Bond albedo for Earth doesn't change much more, as long as it can hold life with the little carbon dioxide left, some boundaries for the *absorption mass*, M_κ, can be set for planets with life in planetary systems similar to our own Solar System.

For planets with life:

$$M_{dead} < M_{life} \leq 8.6\ M_{dead} \tag{10}$$

In Figure 36, curves are plotted for the formulas:

$$M\ (1 - A_b) = M_{dead} \tag{11}$$

$$M\ (1 - A_b) = 8.6\ M_{dead} \tag{12}$$

The inner planets without life is found near the curve for Eq. (11). The values for Earth place it at a long distance on the curve for Eq. (12). The shaded area is a loose estimate for where planets with life would be found in such a diagram.

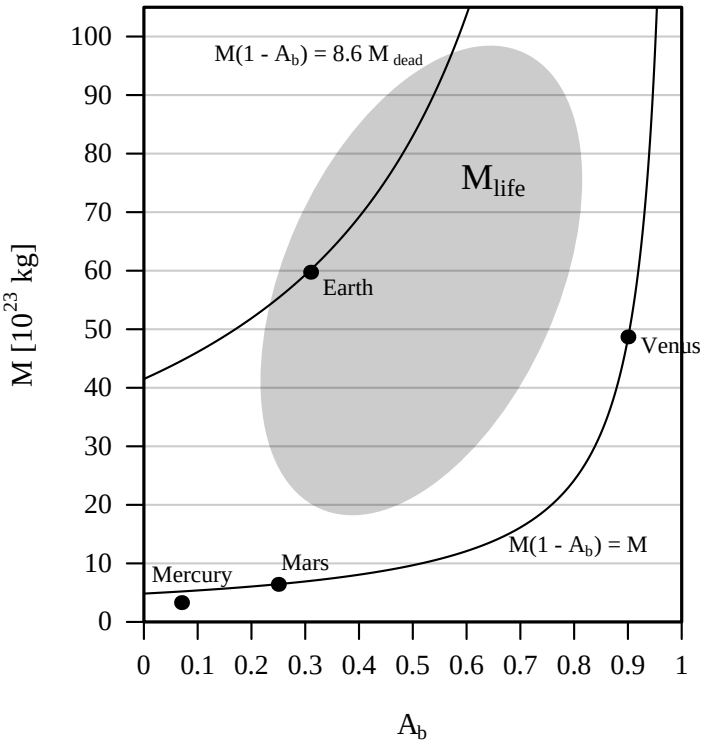

Figure 35: Inner Planets.

Hypothesis: The longer to the right, a planet with life is placed in the diagram, the higher Bond albedo it has, and therefore the heavier atmosphere with more carbon dioxide, and therefore the earlier the state of life is.

Question: Is it possible to set some limits on the mass of and the Bond albedo for planets with possible life from this diagram?

The Dead Mass Constant

The Dead Mass constant was estimated in Eq. (9) as:

$$M_{dead} \approx 4.84 \pm 0.03 \times 10^{23}\ kg$$

This should be thought of as a radiation weighted mass of an Earth-like planet without life in the habitable zone of a star with a luminosity as our Sun.

Planets with life will have radiation weighted masses higher than this constant, up to the radiation weighted mass of the Earth, which is ~ 8.6 times this constant.

This is under the assumption, that the Bond albedo for the Earth can't get much lower than its current value of ~ 0.306, because the atmosphere is so developed, that there is almost no carbon dioxide left.

A Universal Formula

If there is just one planet in the habitable zone around a star, and to be able to directly compare planets from star to star, a universal formula for the *absorption mass*, M_κ, is desired. This can be achieved by including the star luminosity, L, in values of the solar luminosity, $L_\odot$.

$$M_\kappa \equiv \frac{M\ (1 - A_b)\ L_\odot}{L} \tag{13}$$

Now the *dead mass constant*, M_{dead}, estimated in Eq. (9) can be used universally across planetary systems from star to star.

So if a planet with life orbits a **less luminous** star than our Sun at the distance of 1 AU, the planet would need to have a **smaller mass** than the Earth to have an equal clear and developed atmosphere as the Earth.

If on the other hand a planet with life orbits a **more luminous** star than our Sun at the distance of 1 AU, the planet would need to have a **larger mass** than the Earth to have an equal clear and developed atmosphere as the Earth.

Knowing the Effective Temperature

If instead of the Bond albedo, A_b, the effective temperature, T_{eff}, is known, a similar formula can be constructed.

$$\frac{M\ r^2\ T_{eff}^4\ L_\odot}{L} \approx constant \tag{14}$$

Eq. (14) holds for Earth-like planets without life in the habitable zone.

For further explanation of Eq. (14), see *"Spot Life from Planet's Effective Temperature"* (Niclasen, 2013).

G.4 Conclusion

The *dead mass constant* for Earth-like planets without life in the habitable zone was studied and interpretation was given. Universal formulas to calculate the *absorption mass* of all Earth-like planets in the habitable zone was given, both from the Bond albedo and the effective temperature. It is assumed, the planet mass and distance to its star is known.

Knowing the *absorption mass* of such planets can help determine, if there is life or not.

H Lapse Rate Experiment

Lapse Rate Experiment

By John Niclasen

April, 2016

Abstract. This experiment will test if the dry adiabatic lapse rate holds in an isolated system.

H.1 Theory

Dry Adiabatic Lapse Rate

The 1^{st} law of thermodynamics states

$$dQ = dU + dW \tag{15}$$

$$\Leftrightarrow \quad dQ = c_V \; dT + P \; dV \tag{16}$$

If, in an ideal gas, a parcel of air moves *adiabatically*, i.e., no heat is exchanged between the parcel of air and its surroundings ($dQ = 0$), the 1^{st} law of thermodynamics requires that

$$c_V \; dT = -P \; dV \tag{17}$$

$$c_P \; dT = \frac{1}{\rho} \; dP \tag{18}$$

Also assuming *Hydrostatic Equilibrium* means

$$\frac{dP}{dz} = -g\rho \tag{19}$$

Substituting eq. (19) into eq. (18) leads to

$$\frac{dT}{dz} = -\frac{g}{c_P} \tag{20}$$

Eq. (20) is the *Dry Adiabatic Lapse Rate* (DALR), and is often symbolized by the greek letter known as Γ.

$$\Gamma_d \equiv -\frac{dT}{dz} = \frac{g}{c_P} \tag{21}$$

H.2 The Experiment

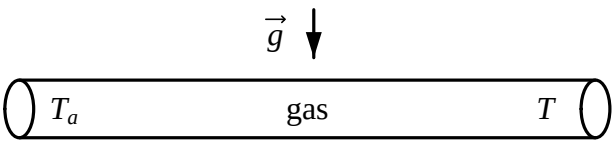

Figure 36: A horizontal tube of gas.

As shown in Fig. 1, a tube is filled with gas. This can be ordinary air or some other gas like carbon dioxide to test different effects with different gasses. The tube is placed in a horizontal position and is well isolated from the surroundings. After some time, the gas will reach an equilibrium temperature, so

$$T_a = T_g \tag{22}$$

Now the tube is rotated 90 degrees:

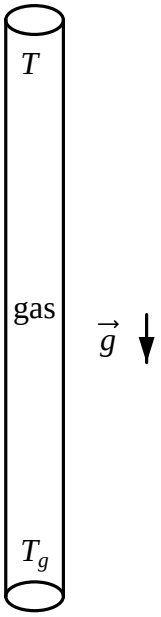

Figure 37: A vertical tube of gas.

If the dry adiabatic lapse rate holds in an isolated system, then when the vertical tube of gas has reached equilibrium, there will be a temperature difference, e.g.

$$T_a < T_g \tag{23}$$

I Matrices

```
World [
    Title:      "Matrix Operations"
    Date:       31-Mar-2016
    Version:    0.1.0
    File:       %matrix.w
    Author:     "John Niclasen"
    Purpose:    {Direct matrix operations using blocks and
        complex! as dimension (should be pair!).}

    Rights:     "Copyright © 2016 John Niclasen, NicomSoft"

    History:    [
        0.1.0   [31-03-2016 JN  {Created.}]
    ]
]


mmul: make function! [[
    "Matrix multiply."
    A B
    /local x y z C i j k v AA BB
][
    x: to integer! A/1/1
    y: to integer! B/1/2
    z: to integer! A/1/2
    C: make block! x
    insert C x * 1+0i + (B/1/2 * 1i)
    AA: next A
    BB: next B
    i: 0
    while [i < x] [
        i: i + 1
        j: 0
        while [j < y] [
            j: j + 1
            k: 0
            v: 0.0
            while [k < z] [
                k: k + 1
                v: (pick AA i - 1 * z + k) * (pick BB k - 1 *
                    y + j) + v
            ]
            append C v
        ]
    ]
    C
]]


mT: make function! [[
    "Matrix Transpose."
    A
    /local B rows cols i j
][
    B: copy []
    rows: to integer! A/1/1
    cols: to integer! A/1/2
    insert B rows * 1i + cols
    j: 0
    while [j < cols] [
        j: j + 1
        i: 0
        while [i < rows] [
            append B pick A i * cols + j + 1
            i: i + 1
        ]
    ]
    B
]]
```

```
mdet: make function! [[
    "Matrix Determinant, only for 2x2 and 3x3 matrices so far
        ."
    A
][
    ; A/1 is dimension, m+ni
    either A/1 = 2+2i [
        A/2 * A/5 - (A/3 * A/4)
    ][
        A/2 * A/6 * A/10 - (A/2 * A/7 * A/9) - (A/3 * A/5 * A
            /10)
        + (A/3 * A/7 * A/8) + (A/4 * A/5 * A/9) - (A/4 * A/6
            * A/8)
    ]
]]


minv: make function! [[
    "Inverse matrix, A^(-1), only for 2x2 and 3x3 matrices so
         far."
    A
    /local C d
][
    C: copy []
    insert C A/1
    d: 1 / mdet A
    either A/1 = 2+2i [
        append C reduce [d * A/5 negate d * A/3 negate d * A
            /4 d * A/2]
    ][  ; 3+3i matrices
        append C reduce [
            d * mdet reduce [2+2i A/6 A/7 A/9 A/10]
            d * mdet reduce [2+2i A/4 A/3 A/10 A/9]
            d * mdet reduce [2+2i A/3 A/4 A/6 A/7]
            d * mdet reduce [2+2i A/7 A/5 A/10 A/8]
            d * mdet reduce [2+2i A/2 A/4 A/8 A/10]
            d * mdet reduce [2+2i A/4 A/2 A/7 A/5]
            d * mdet reduce [2+2i A/5 A/6 A/8 A/9]
            d * mdet reduce [2+2i A/3 A/2 A/9 A/8]
            d * mdet reduce [2+2i A/2 A/3 A/5 A/6]
        ]
    ]
    C
]]


mdiag: make function! [[
    "Make diagonal matrix."
    v
    /local D l i j
][
    l: -1 + length? v
    D: copy []
    insert D l * 1i + l
    i: 0
    while [i < l] [
        i: i + 1
        j: 1
        while [j < i] [
            j: j + 1
            append D 0.0
        ]
        append D v/(i + 1)
        j: i
        while [j < l] [
            j: j + 1
            append D 0.0
        ]
    ]
    D
]]
```

```
mprint: make function! [[
    "Print matrix."
    M
    /local x y i j
][
    x: to integer! M/1/1
    y: to integer! M/1/2
    next' M
    i: 0
    while [i < x] [
        j: 0
        prin "    "
        while [j < y] [
            j: j + 1
            prin pick M i * y + j
            prin " "
        ]
        prin "^/"
        i: i + 1
    ]
    head' M
]]
```