Development of an Irradiance Sensor Based on a Photon Counting Camera

Lars Lønne

Master of Science in Engineering and ICT
Submission date: December 2014
Supervisor: Asgeir Johan Sørensen, IMT

Norwegian University of Science and Technology
Department of Marine Technology

NTNU Trondheim
Norwegian University of Science and Technology
Department of Marine Technology

MASTER THESIS IN MARINE CYBERNETICS

FALL 2014

FOR

STUD. TECH. Lars Lønne

Development of a Prototype Sensor for Measuring Irradiance in the Polar Night

Work description

In January 2014, the Marine Night project conducted a research campaign in Ny-Ålesund. As part of this campaign, a new prototype sensor was tested. The sensor measures minuscule amounts of light, such as are found in the Arctic polar night. This light might be important for life in the Arctic, and researchers need tools such as this sensor to measure it.

The initial test of the sensor was promising. Now, for the researchers to be able to fully use this new sensor there exists a need for a more complete software package than what was developed for the previous campaign.

The aim of this thesis is to develop a software package that will enable researchers to use this sensor for measuring light in the Arctic polar night. This work is a continuation of the candidate’s previous work. This master thesis should consist of two scientific papers, with a résumé. The first paper should look at the results of deploying the sensor during the 2014 campaign and the implications of those results. The second paper should detail testing and verification of the sensor.

Scope of work

Résumé

- Discuss the need for a new sensor, and its importance to ongoing Arctic research.
- Describe the architecture and design of the software package.

Paper 1

- Review the theoretical background for light, its characteristics.
- Review the need for irradiance measurements.
- Describe the experiment carried out during the 2014 campaign.
- Present the results from the campaign.
- Discuss the results, and their importance for further work.

Paper 2

- Describe the new sensor structure and implementation.
- Describe experiments for testing the correctness of the new sensor.
- Discuss the results of the experiments, and provide suggestions for further work.

The report shall be written in English and edited as an article collection with a résumé in front, in the format of a report. It is supposed that Department of Marine Technology, NTNU, can use the results freely in its research work, unless otherwise agreed upon, by referring to the student's work. The thesis should be submitted by 08 February, 2015.

Co-supervisor: Professor Geir Johnsen

Professor Asgeir J. Sørensen
Supervisor

Abstract

This thesis describes the development and testing of a new irradiance sensor. Light from the sun affects all life on earth, and recent studies give reason to believe that some life might even react to the microscopic levels of light found in the Arctic polar night. Commercial light meters are not sensitive enough to register the light in these dark periods of the year, thus the need for a new sensor has emerged.

During a field campaign in Ny-Ålesund in January 2014, conducted as part of the Marine Night Research project, a prototype of the sensor was first tested. The results were promising, and work began to develop a more complete software system that is able to measure the light environment of the Arctic for extended periods of time. The development of this sensor system, and initial tests, are described in this thesis.

This part of the project ended with a working system that is to be tested on the next field campaign of the Marine Night project, in January 2015. A manual has been written for the users, and is included as part of this thesis. Also included is the code documentation, intended for the future developers of the system.

Sammendrag

Denne masteroppgaven beskriver utviklingen og testingen av en ny type irradianssensor. Sollyset påvirker alt liv på jorden, og nyere forskningsresultater tyder på at noen livsformer kan være avhengig av de mikroskopiske lysnivåene i den arktiske vinteren. Forskerene som studerer disse livsformene, trenger en ny type lysmåler, siden målerene som er tilgjengelig i dag ikke er følsomme nok for denne mørke tiden på året.

I forbindelse med et forskningstokt i Ny-Ålesund i januar 2014, ble en prototype av denne nye sensoren testet. Toktet ble utført som en del av forskningsprosjektet Marine Night. Resultatene var lovende, og arbeidet begynte dermed for å utvikle et mer komplett system. Dette nye systemet skal kunne måle de arktiske lysforholdene gjennom lange perioder av året. Utviklingen av dette sensorsystemet og de første testene er beskrevet i denne oppgaven.

Dette prosjektet endte med et system som tilfredsstiller de primære kravene fra forskerne. Det skal testes på det neste toktet i regi av Marine Night, i Ny-Ålesund i januar 2015. En brukermanual har blitt skrevet for dette toktet, og er vedlagt som en del av oppgaven. Dokumentasjon av vesentlige deler av koden er også vedlagt, og skal være til hjelp for framtidige utviklere av systemet.

Preface

This master thesis is written during the fall semester of 2014 as the final part of the master program *Engineering and ICT* at the department of Marine Technology at the Norwegian University of Science and Technology (NTNU) in Trondheim.

This thesis is written in close cooperation with researchers from the *Marine Night* research project. *Marine Night* is part of the larger *Mare Incognitum* research project, registered as project number 226417 with The Research Council of Norway.

The thesis is edited as a collection of papers with a résumé in front. The first paper is titled *Development of a Prototype Sensor for Measuring Irradiance in the Polar Night*, and the second paper is titled *Testing and Verification of a Low Light Irradiance Sensor*.

Acknowledgements

I would like to thank my supervisor Professor Asgeir J. Sørensen and my co-supervisor Professor Geir Johnsen for their contribution to this thesis. Asgeir has been an excellent guide in this project, and is always a source of great advice. Geir has always been enthusiastic about the project, and has provided a number of good ideas along the way. They have been a joy to work with.

I would also like to thank my colleagues at Kantega in Trondheim. While working there I have learned a great deal about software development, and web development in particular. That knowledge has proved invaluable for this project.

A special thanks goes to Inga and Sigrid, who makes it all worthwhile.

Contents

1 Introduction

1.1 The need for a new sensor

Until recently, it was a widely held belief that biological processes in the Arctic slows down or stops during the polar night. In Berge et al. [2009], the opposite is shown to be true. Using acoustic measurements, the process known as Diel Vertical Migration (DVM) is shown to persist through the Arctic winter. DVM represents a huge movement of biomass, and is of great importance to the Arctic marine ecosystem.

Further, Berge et al. [2009] argues that DVM in the Arctic polar night is controlled by variations in solar and lunar light intensity. Light intensities in the polar night are miniscule, and well below the intensities where the human eye is effective. To study the effects of the light in the polar night, the intensity of the light must be reliably measured. Such low intensities calls for a sensor which is much more sensitive than what is generally available in the market today, thus it must be custom made.

A camera called the Single Photon Counting Camera (SPC2), made by Micro Photon Devices[1], was chosen as a new light measurement instrument. It is a precision instrument, capable of single photon detection with an efficiency from 15% to over 40% in the 400 nm to 700 nm band. The sensitivity of this camera will enable researchers to quantify the light levels of the Arctic polar night, and further investigate the biological processes during winter in the high Arctic.

1.2 Development of the new sensor

The SPC2 provides the possibility of measuring tiny intensities of light, but it is not delivered with suitable software for this purpose. The company provides a Software Developer Kit (SDK) and a test application, which is able to capture images and try out the camera. This project needed to capture time series of light measurements; for this purpose, the test application fell short. Thus it was necessary to build a custom made software package.

The new light sensor was developed for researchers in marine biology, associated with the research program *Marine Night*[2]. Together with Professor

[1]http://www.micro-photon-devices.com/
[2]http://www.marinenight.mare-incognitum.no

Geir Johnsen[3], a detailed specification of the software system was drawn up. This specification is included in Appendix B.

1.3 Structure of the thesis

This thesis is presented as a collection of two papers with a résumé in front. The purpose of the résumé is to collect and present background theory and concepts in one place, leaving the papers to concentrate more on the results.

Section 2 describes the developed software system. The software architecture is presented and explained, along with some methods on software development. Also described is the theory and implementation of robust data storage.

Section 3 presents some possibilities for further development of the sensor. This section contains ideas and plans for future use, especially concerning use of the sensor together with underwater platforms.

Paper 1 is called *Development of a Prototype Sensor for Measuring Irradiance in the Polar Night*, and describes the first testing and deployment of the sensor during the January 2014 *Marine Night* field campaign.

Paper 2 is called *Testing and Verification of a Low Light Irradiance Sensor.* It details several experiments that were performed to determine the new sensor's correct operation. The tests were simple, but illustrates that the sensor is giving output in the correct range, and that it is capable of capturing the dynamics of changing light conditions.

The appendices include the user manual, software specification and the code documentation. The user manual is intended for the end users of the software, while the code documentation is intended for future developers and maintainers of the software modules. There is some repetition in the texts in the appendices, especially in the user manual and the code documentation. This is intentional, so that the manuals can be read independently of the thesis.

A zip archive accompanying this thesis includes the source code for the software system.

[3]Professor Geir Johnsen, Department of Biology, NTNU

2 Software system

This section describes the software package that was developed as part of this thesis. The software package has been developed using the test-driven development technique. Subsection 2.1 discusses test-driven development, and the *test first* concept. Subsection 2.2 describes the software specification, and discusses the its importance. Subsection 2.4 presents the software architecture, and details the ideas behind it, especially the decision to separate the server and web applications into two independent modules. Finally, subsection 2.5 discusses some theory on robust storage, and demonstrates how this is implemented in the server application.

2.1 Development concepts

The software was developed following the guidelines of test-driven development. This development technique increases confidence in the correctness of the program, and has also been found to increase productivity [Erdogmus, 2005].

2.1.1 Test-driven development

Test-Driven Development (TDD) is a technique based on writing functional tests for a piece of production code *before* writing the code itself. A typical TDD workflow is as follows:

1. *Writing the test.* The first step in TDD is to read and understand the requirements, and write a functional test. The test is often written in a unit testing framework, such as JUnit[4] if one is developing Java applications, or Mocha[5] for the node.js platform.

   The requirements must be described in a clear, concise manner, with as little ambiguity as possible. This is the job of the *specification*, which will be discussed further in subsection 2.2. Writing the test first forces the developer to really understand the requirements beforehand.

2. *Run all tests.* This step should conclude with all previous tests succeeding, and the new test failing. This confirms that the new test is adding new information to the system, in the sense that it requires

[4]http://junit.org
[5]http://mochajs.org

new code to pass. It also confirms for the developer that the testing platform works, and that the new test is actually looking for the new functionality.

3. *Write production code.* New code is now added to the code base. Only the minimal code needed to make the test pass is added at this stage.
4. *Re-run tests.* All tests are now run again, to make sure that the new test passes. If it does not, the developer goes back to the previous step. This cycle is repeated until all tests pass.
5. *Refactoring.* As the code base grows larger, refactoring becomes increasingly important. Duplication must be removed, code can be moved to where it more logically belongs, and code can be rewritten to increase readability or maintainability. At this point, all functionality is covered by the functional tests. After refactoring, all tests can be run again to make sure that the refactoring has not introduced any new bugs.

According to a study from the National Research Council of Canada [Erdogmus, 2005], following the TDD technique leads developers to write more tests, be more productive and it raises the minimum quality of the finished software. It also increases confidence in the correctness of the program, which is especially important for a long running process such as this.

2.2 Software specification

A software specification is a detailed list of the customer's requirements for the software system. The specification is a crucial part of the development process, because the specification document lists the requirements for the system. Both functional and non-functional requirements should be documented. Functional requirements specifies how the program should work. Non-functional requirements specify other requirements to the software, such as uptime requirements for a server. It is vital that both the customer and the developer fully understands and adheres to the specification. Any changes to the requirements, either functional or non-functional, should first be listed in the specification.

A detailed specification was developed together with Prof. Geir Johnsen. The final specification is included in Appendix B. The document is divided first into *Software functionality* and *Hardware functionality*, then into subgroups under these headings. Each subgroup, such as *Capturing images* and *Storing data*, is further divided into priority groups. *Priority A* are requirements for a functional system. These have top priority. *Priority B* tasks are not absolutely required, but functionality that the customer would like to see in the system. *Priority C* tasks are tasks meant for future development, and have lowest priority.

The specifications were consciously made with too many tasks for this development cycle. Thus it also lists future adaptations of the system that are not realized at this point.

2.3 Selecting a language and framework for the server

2.3.1 Requirements

The server must communicate with the SPC2 sensor. Communication and control of the sensor is done through a C library. Hence, the server must be able to call C functions.

Writing a server program from scratch is time consuming and unnecessary. Frameworks exist that will handle the low level details, allowing developers to focus on the parts that are unique to their implementation. The choice of language should have a good framework that will make development easy and efficient.

To enable the client to communicate with the server, and indirectly with the sensor, the server must expose some kind of API (Application Programming Interface). This server will expose a REST API, which uses an internet media type (often JSON) to transfer data through standard HTTP methods, such as GET, POST, PUT and DELETE. The client will then make AJAX calls to query the server and control the sensor. AJAX is an acronym for *Asynchronous Javascript + XML* and is the standard technique to make asynchronous calls to an API on the client side. Nowadays, JSON is often favored over XML as the data format language of choice. The choice of programming language and framework should facilitate the creation of this API.

The API must be secured against unauthorized use and attacks. For this application, basic access authentication[6] will suffice. The choice of language and framework should enable the developer to easily implement this authentication scheme.

[6]https://developer.mozilla.org/en-US/docs/Web/HTTP/Basic_access_authentication

2.3.2 Candidate: C++

The server has to make calls to a C library, and this is most easily done from C/C++. Thus, one candidate programming language for the server is C++.

Writing a REST API in C++ is not common. The language is more often used for applications where either hardware access or speed of computation are top priority. Thus, there are not many server frameworks written in C++. One exception is Microsoft's C++ REST SDK (codename "Casablanca")[7].

2.3.3 Candidate: Java/Spring

Spring is a framework for Java EE. From the spring webpage[8]:

> *Spring helps development teams everywhere build simple, portable, fast and flexible JVM-based systems and applications.*

Java is, in the author's opinion, easier to use than C++. Spring enables rapid development of a REST API, through the use of Java annotations and the *Model-View-Controller* software architectural pattern. Basic authentication is also available through the use of the *BasicAuthenticationFilter* class.

To call the C library from the SPC2 SDK, Java makes use of JNI, the *Java Native Library.* Using the JNI with a precompiled library, requires three steps:

1. Write a declaration in Java for the library function you want to call. This declaration must use the `native` keyword, to specify that the code is written in some native language, in this case C.
2. Create a C header file for use by the native code. This is done automatically with the `javah` tool.
3. Implement the native method in C. This requires the developer to write another C program which is a wrapper around the library, and which includes the header created in step 2.

2.3.4 Candidate: node.js and express

From the node.js webpage[9]:

[7]https://casablanca.codeplex.com/
[8]http://spring.io
[9]http://nodejs.org

Node.js is a platform built on Google Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Express[10] is an addon to node.js. It is a

[...] minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications.

Together, node.js and express makes the job of creating a REST API and securing it with basic authentication very easy. Node.js is made for building network applications, and includes many time saving modules, such as `http`. `http` is a module that

[...] supports many features of the protocol which have been traditionally difficult to use.

For calling code from the C library, node.js has an addon called `node-ffi`; the Node.js Foreign Function Interface. It enables node.js developers to load and call dynamic libraries using pure JavaScript. `node-ffi` also handles the conversion of types between JavaScript and C.

2.3.5 Conclusion

The server will be developed with node.js and Express. Node.js appears to offer the easiest and most time saving solution, being purposely built for developing network applications. Also, the `node-ffi` module appears much easier to use than the Java Native Interface, which will save much development time.

Java and Spring was not chosen, mostly because of its complexity. While it is perfectly feasible to write this program with Spring, node.js seems to present a much cleaner, easier solution without much of the overhead. For a simple project such as this, Spring seems to complex.

C++ was not chosen for three reasons. First, it appears to be much too complex for this simple project. Second, the lack of frameworks seem to indicate that C++ is not much used for this kind of development. Third, the developer does not have much experience with C++, and using it for this

[10]http://expressjs.com

project would require some time studying the language itself. That time is better spent on developing the actual program.

2.4 Architecture of the server application

The architecture of the server application is visualized in figure 1.

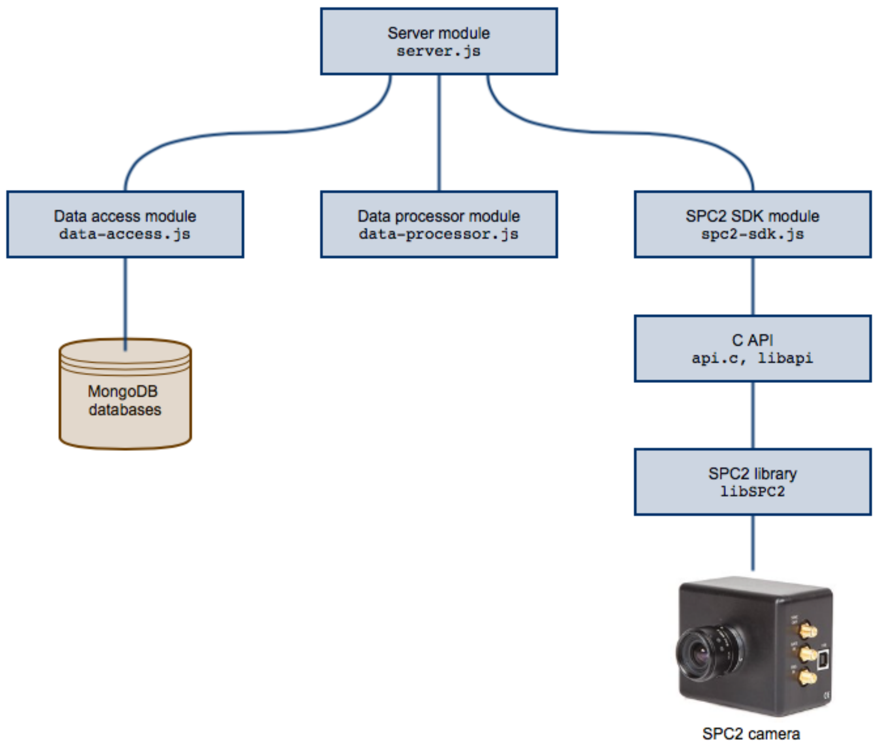


Figure 1: The architecture of the server application.

The design of the applications modules follows the *Single Responsibility Principle*, a term coined by Robert C. Martin [Martin, 2002]. The principle states that *A class should have only one reason to change.* Extending this from classes to modules, each of the modules in the server application should have only one responsibility. Thus, the module `data-access` handles all communication with the databases, `data-processor` collects functions which processes data in some way, e.g. calculating the number of photons per second from the raw data. The `spc2-sdk` and `c-api` modules handles communication with the camera, on different levels of abstraction. The `c-api` module is written in C, and communicates directly with the camera's library of functions. It exists to abstract away C pointers from the node.js application. The `spc2-sdk` module exists to translate the C function calls into javascript.

The *server module* contains the top-level logic for the entire application. It defines the Application Programmer Interface (API) that enables other application to communicate with the SPC2 camera. The API is built as a REST API, which means that all communication with the API is done with standard HTTP request methods, such as `GET` and `POST`. It is built as an Express application [Express.js, 2014], and handles the routing of requests from the clients to the other modules.

An important feature of the architecture is that there is no cross-communication between modules on the same level of abstraction. This is intentional, and assures that each model can be used as a black box abstraction. Thus, each module can be extended and maintained independently of the others. As an example, if in the future there is a desire to change the database system to the latest and greatest database, one need only rewrite the `data-access` module. It does not depend on any of the other modules, and can be replaced by another, assuming that the new module provides the same functions as the existing.

2.4.1 Separation of server and web application

The server application and the web application have been developed as two separate modules. This decision increases the complexity, as the two could have been merged and created as one web application.

The reason for the separation is future use. For this project, it is assumed that the camera will be used to create time series of light measurements, and remote access through a web application is a reasonable way to communicate with the sensor.

In the future, it is very likely that the camera will be part of an underwater sensor platform, either moored or mounted on an underwater vehicle. In this case, it may still be desirable to use the server/client architecture, but not with a web application client. The server code can then easily be reused, or modified for new use, without concern for the web application part.

2.5 Robust storage

This section will present some theory on the robust storage, before detailing the way data storage is implemented in the server application. The theory in this section is based on Burns and Wellings [2009].

Two approaches that can help designers improve the reliability of their systems are *fault prevention* and *fault tolerance*. Prevention attempts to eliminate sources of faults. Tolerance enables the system to continue functioning in the presence of faults.

2.5.1 Fault prevention

There exists two stages of fault prevention; fault avoidance and fault removal. Avoidance tries to limit the introduction of potentially faulty components during the construction of the system. For hardware, avoidance includes the following strategies:

- Use the most reliable components.
- Use refined techniques for interconnecting components and assembling subsystems.
- Package the hardware to eliminate interference.

It is virtually impossible to write fault free code. Even bugs that are very unlikely to occur can come to the surface in a long running program. To improve the quality of software, developers can

- rigorously specify the requirements.
- use proven design methodologies.
- use analysis tools to verify properties of the software.
- use languages that facilitate modular structure and abstractions.

No matter how careful the developers are, faults will inevitably be present in the system after its construction. Hence, the second stage of prevention is fault removal. Fault removal consists of procedures for finding and removing bugs. A ubiquitous example of fault removal is system testing, in all its forms. Unfortunately, testing can almost never be exhaustive, due to the complexity of most larger software systems.

- A test can only prove the presence of faults, not their absence.

- It may be impossible to test under realistic circumstances.
- Errors may not manifest themselves before the system goes operational.

In spite of all testing and verification, hardware components will fail. Fault prevention is therefore insufficient if the system must continue operating without the aid of maintenance personnel.

2.5.2 Fault tolerance

In many ways, fault tolerance is the counter example to fault avoidance. While avoidance focuses on not experiencing faults in the first place, fault tolerance assumes that faults will be present, and implements strategies to continue operating despite this fact. It is important that fault tolerant systems caters for unforeseen faults, as well as the anticipated ones. Fault tolerance can be implemented on three levels of complexity:

Full fault tolerance The system continues to operate in the presence of faults, albeit for a limited time period. It experiences no significant loss of functionality or performance. This state of operation is not possible to sustain indefinitely.

Graceful degradation Also called fail soft. The system continues to operate, accepting a partial degradation of functionality or performance.

Fail safe The system saves its state and comes to a halt.

2.5.3 Redundancy

Redundancy is a common way to implement fault tolerance. Protective redundancy is adding extra elements to the system, enabling it to detect and recover from faults. While redundancy increases the fault tolerance of the system, it also increases the complexity. The aim then, must be to minimize redundancy, while at the same time maximizing reliability.

One specific implementation of redundant fault tolerance is static redundancy. Here, faulty components are masked out, or hidden from the system. As an example, in *Triple Modular Redundancy*, three identical components work in parallel, and a voting circuit masks out an eventual faulty component. This can be extended with more components, and is in the general case called *N Modular Redundancy*.

Dynamic redundancy is code running inside a component, which tests the component and indicates the system if the component is faulty. It provides no redundancy in itself, in the sense that another component must take over for the faulty one.

2.5.4 Redundancy implementation in the server application

To achieve robust storage in the server application, it makes use of a feature in MongoDB called *replication*. The documentation [MongoDB, 2014] states:

> *A replica set in MongoDB is a group of* `mongod` *processes that maintain the same data set. Replica sets provide redundancy and high availability, and are the basis for all production deployments.*

Hence, MongoDB has static redundancy built in, albeit under a different name. Replication works with a set of `mongod` processes, `mongod` being the process that runs an instance of the database system. One of the processes is named the *primary* and receives all write commands from the client. The client in this case being the SPC2 server application. Every other `mongod` process is called a *secondary*. Every secondary process ensures that they have the same data set as the primary by applying the primary's operations.

To maintain the set of database processes, each member of the replica set sends *heartbeats* to all other members every two seconds. If one member is not heard from in 10 seconds, it is assumed offline. Then, the remaining members elect a new primary among themselves.

In the server application, the addresses of database processes is entered into the config file. On startup, the application, through the `mongoose` module [Mongoose.js, 2014], will connect to the replica set, and provide seamless static redundancy. The application assumes that the `mongod` processes are configured and belongs to the same replica set. This process is detailed in the user manual in appendix A.

2.6 User acceptance test

On 24 November 2014, an acceptance test was performed with the user[11]. The software was presented, along with a table of the priority A requirements from the specification. The results are shown in table 1.

[11]The user in this context is Prof. Geir Johnsen, Department of Biology, NTNU.

Functionality	**Accepted?**
Set and change integration time and dead-time correction	Yes
Save information about camera lens.	Yes
Reset parameters to default values.	No
Store values for all parameters.	Yes
Capture images in automatic mode.	Yes
Store data in a reliable and secure manner.	Yes
Each image, or equivalent data, must be stored.	Yes
Store result in quantum scale.	No
Parameters must be stored with the data.	Yes
Calibration factor must be stored with the data.	Yes
User access to stored data.	Yes
Download entire database.	No
Display captured data as plot in web interface.	Yes
Remote access to the sensor, both read and write access.	Yes
Secured against unauthorized access.	No

Table 1: Results from the acceptance test.

Overall, the user was satisfied with the product. There were a couple of points that needed to be addressed before delivery.

- The functionality to reset parameters to their default values was dropped. The function was not yet implemented, but the user did not think that the reset functionality had much value. Parameters will have to be adjusted for every experiment, and it would be very difficult to find good defaults.
- In addition to the values that were already stored in the database, the user would like to also record temperature. The camera does not have a temperature sensor, and the logging of temperature were not in the original specifications. It was decided that temperature logging would not be implemented in this version.
- When examining the quantum scale output, the numbers seemed to be too low. The fault would need to be found and corrected.
- Downloading the entire database was not implemented, but the user decided to drop it. The database files can be downloaded over `ssh` if needed.
- Security was not implemented yet, thus it was not accepted.

2.6.1 Delivery

The corrections from the acceptance test were made, and the software system was delivered and accepted on 15 December 2014.

3 Future adaptations

3.1 Underwater operations

When the sensor has been thoroughly tested on land, the next step will be to further develop it for use underwater. Accurate measurements of the underwater light environment is crucial for further research on marine life in the polar night. In Berge et al. [2012], the activities of bioluminescent zooplankton in the Arctic is described. The zooplankton is shown to follow a diel vertical migration pattern, that does not follow any detectable changes in illumination. This suggests that the zooplankton either react to light that is too low for current sensor solutions to register, or that the levels of light are so low that other mechanisms must be driving these vertical migrations. A better understanding of the available light in the underwater environment could help drive this line of research forward.

The SPC2 is capable of high speed captures at up to 48 000 kHz. This could be utilized to measure light dynamics, e.g. of the bioluminescent zooplankton described in Berge et al. [2012]. Continuous measurements at high speed could give a level of detail in the light measurements that is not possible with slower or less sensitive sensors.

For the sensor to be usable in underwater missions, a number of modifications and further developments will have to be done. Most importantly, the sensor needs an underwater housing. The camera is not built for underwater use, thus a custom housing will need to be built. The housing must have room for the camera itself, a computer for controlling the camera, a logging unit for storing the collected data, and a power supply. The sensor might be deployed on a number of different platforms, such as Autonomous Underwater Vehicles (AUV), Remotely Operated Vehicles (ROV) or stationary platforms, such as a moored sensor station. The housing will need to take into account the restraints of the platform, such as weight allowances, geometric shape, and possibly hydrodynamic efficiency in the case of the AUV.

The sensor's software will have to be further developed for underwater use. At this point, the software is designed for prolonged use in a safe environment indoors. The focus has been on reliability and ease of use. For an underwater operation, the specification will change in a major way. Reliability will be important, perhaps more so. The sensor must continue working, even when failures are present. The sensor must be autonomous, such that it can be started at the beginning of the mission, and left running without user input until the platform it is deployed on is recovered. Lastly, if the sensor needs to interface with the platform's systems, special interface code might be called for.

3.2 Power system

The camera is powered by a 5 V, 2.4 A power source, resulting in a 12 W power requirement. Power for the controlling computer is also required, and is typically around 60 W.

In an indoors environment, for relatively short periods of time, power supply reliability is usually not a problem. For the sensor's future intended use, where it is expected to reliably measure the available light for months on end, reliable power can be an issue. Especially if the sensor is deployed in a location where maintenance is hard to come by, the power supply must always be available.

A constant power supply can be obtained with a battery. The battery must be sized to provide the sensor with power for longer than the worst case outage, within reason. A further development of the sensor would be to investigate the possibilities of an outage and calculate the probable worst case scenario. A battery package could then be designed to power the sensor, while the battery itself is charged as long as there is available power.

References

J. Berge, A.S. Båtnes, G. Johnsen, S.M. Blackwell, and M.A. Moline. Bioluminescence in the high arctic during the polar night. *Marine Biology*, 159 (1):231–237, 2012. ISSN 0025-3162. doi: 10.1007/s00227-011-1798-0. URL `http://dx.doi.org/10.1007/s00227-011-1798-0`.

Jorgen Berge, Finlo Cottier, Kim S Last, Oystein Varpe, Eva Leu, Janne Soreide, Ketil Eiane, Stig Falk-Petersen, Kate Willis, Henrik Nygard, Daniel Vogedes, Colin Griffiths, Geir Johnsen, Dag Lorentzen, and Andrew S Brierley. Diel vertical migration of Arctic zooplankton during the polar night. *Biology Letters*, 5(1):69–72, 2009.

Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*. Addison-Wesley Educational Publishers Inc, USA, 4th edition, 2009. ISBN 0321417453, 9780321417459.

Hakan Erdogmus. On the effectiveness of test-first approach to programming. In *Proceedings of the IEEE Transactions on Software Engineering, 31(1)*, 2005.

Express.js. Express - node.js web application framework, December 2014. URL `http://expressjs.com`.

Robert C Martin. The single responsibility principle. *The Principles, Patterns, and Practices of Agile Software Development*, pages 149–154, 2002.

MongoDB. MongoDB, December 2014. URL `http://www.mongodb.org/`.

Mongoose.js. Mongoose odm v3.8.20, December 2014. URL `http://mongoosejs.com`.

Development of a Prototype Sensor for Measuring Irradiance in the Polar Night

Lars Lønne [*]

[*] *Norwegian University of Science and Technology, Trondheim, Norway. (e-mail: larslonn@stud.ntnu.no)*

Abstract: This report describes the process of developing a prototype sensor for measuring the low levels of irradiance in the Arctic polar night. A highly sensitive photon imager, capable of single photon detection, was used for this purpose. The special purpose software developed for the prototype is also described. The prototype was deployed at 78°55′ N during the Marine Night research campaign. Experimental results are presented and discussed, along with results from another prototype, for comparison.

Keywords: Prototype sensor, Polar night, Irradiance

1. INTRODUCTION

The purpose of this project was to develop a prototype for measuring the low levels of light in the polar night. This project report details the different aspects of this project, such as description of the prototype hardware and development of the required software. Also included are experimental results and suggestions for further work.

This project was done as part of the Marine Night project [1]. Marine Night aims to increase our understanding about the marine biological processes during the polar night, and the light environment is a crucial part of this. Recently discovered marine biological processes (Berge et al., 2009) are among the motivating factors for this research project. One of the hypotheses is that the organisms are able to react to light changes in the polar night, which seem insignificant to the human eye.

Experiments with the prototype light sensor were carried out during the Marine Night field campaign in Ny-Ålesund from 13 to 28 January, 2014. Ny-Ålesund is the world's northernmost permanent settlement, and with its position at 78°55′ N 11°56′ E, it is perfectly situated for studies of the polar night. The purpose of the campaign was twofold: to carry out the research activities of the Marine Night project, and to educate students taking part in the course *AB-334: Underwater Robotics in the Arctic Polar Night* at the University Centre in Svalbard. The author was enrolled in said course.

The paper is organized as follows: section 2 presents the theory and characteristics of light, and its importance for life in the Arctic; section 3 presents the prototype sensor and the software that was developed for it; and section 4 presents the results from the Marine Night campaign.

[1] `http://www.mare-incognitum.no`

2. THEORY

2.1 The polar night

Due to the inclination of the earth, the Arctic regions experience polar night during the winter season. The polar night is a period of the year in which the sun never rises above the horizon. For a few hours at midday, there is some available light due to scattering of the sunlight in the atmosphere (Sakshaug et al., 2009), but for most of the day the environment is seemingly completely dark.

In Ny-Ålesund, the sun sets on 25 October and does not rise again until 17 February. This has a major effect on life in the Arctic. Primary production halts during the winter months, and until recently it was believed that most all life came to a stop during winter. Recent observations, however, has shown that this is not the case (Berge et al., 2009).

2.2 The physics of visible light

The following subsection is based on Young and Freedman (2012).

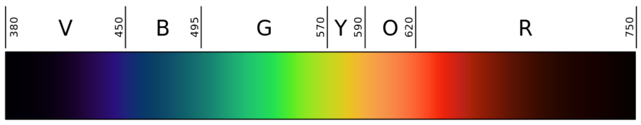


Fig. 1. Illustration of the visible spectrum

Visible light is electromagnetic radiation with a wavelength (λ) in the range 400 nm to 700 nm. Radiation with either shorter or longer wavelengths are invisible to the human eye. The color of the visible light is a function of its wavelength, see figure 1. Visible light ranges from violet with a wavelength around 400 nm, through blue, green, yellow and orange to red, which has a wavelength in the range 620 nm to 700 nm.

The wavelengths of visible light also coincides with the wavelengths of *Photosynthetically Available Radiation* (PAR), which is discussed in section 2.4.

In the above description, light is assumed to behave like a wave, with a given wavelength and frequency. In some situations, such as when describing colors, this model works very well. However, light can also be modeled as discrete particles, called *photons*. Modern physics actually sees light as something which behaves both like a wave and like a particle. This is called the *wave-particle duality*. The manifestation of light either as particles or as waves depend on the mode of measurement; it is not possible to observe both modes at the same time.

In the particle mode, light is characterized by the energy of each particle, and the particle flux. Photons have no inherent color, but when they reach the eye, the energy of the photon is translated by the brain into colors. Brightness is associated with the photon flux; a higher flux corresponds to a brighter source of light. The wave model and the particle model is connected according to Planck's law:

$$e = \hbar c/\lambda$$

e is the energy, $\hbar = 6.62 \times 10^{-34}$ J s is Planck's constant, and c is the speed of light in a vacuum. From Planck's law, it is seen that the energy is inversely proportional to the wavelength. From this, it can be deduced that a photon in the blue band of the spectrum has an inherent energy that is roughly 1.6 times greater than a photon in the red band.

Photons in the visible band all carry enough energy to bring about photosynthesis, and it is the number of photons that matter, not the total energy. Therefore, the particle model is well suited when studying the effect of light on biological life (Sakshaug et al., 2009).

The photon counter described in section 3 measures the incoming light as photons.

2.3 The characteristics of light

Light can be characterized by its intensity, spectral characteristics, polarization and direction.

The intensity of light is measured as irradiance (E). Irradiance is measured either on a quantum scale, with units µmol photons/m^{2}s, or on an energy scale, with units µW/m^2. The irradiance value is a function of the direction. This is called the cosine effect, see section 2.5.

Irradiance is often specified as either downwelling irradiance (E_d) or upwelling irradiance (E_u). Downwelling irradiance is caused by light coming from above, such as sunlight hitting the ocean surface. Upwelling is reflected light coming from below, such as sunlight reflected off the seafloor in shallow waters. The sum of irradiance from all directions in one point is called *scalar irradiance* (E_0). Scalar irradiance most closely resembles what is seen by marine organisms (Sakshaug et al., 2009), and is therefore the preferred unit of measurement for these purposes.

The spectral composition of light quantifies how much energy there is in each waveband on the visible spectrum. Figure 2 shows an example of how the spectral composition of sunlight varies with the sun's altitude. The sun's altitude is defined as the angle between the horizon and the center of the sun's disk.

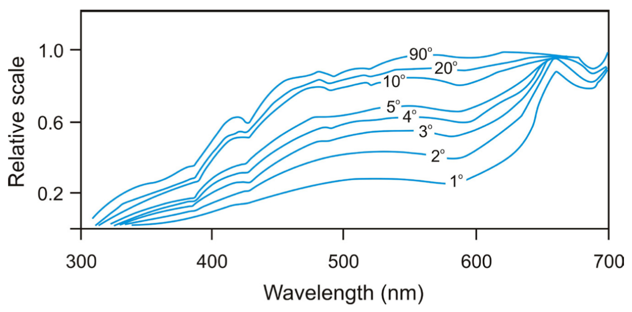


Fig. 2. Spectral composition of sunlight at different solar altitudes. Taken from Sakshaug et al. (2009).

On the Marine Night campaign, the focus was to investigate wether there is discernible differences in the available light throughout the day during the polar night. Hence, irradiance was the most important characteristic. Irradiance was measured with the SPC2 photon imager and the IMO sensor, see sections 4.1 and 4.2.

2.4 The biological importance of light

The information in this subsection is taken from Sakshaug et al. (2009).

Light from the sun is a prerequisite for all life on earth. This light is electromagnetic radiation with a wavelength from 10×10^{-14} m to 2000 km. The human eye can register light waves in the band 400 nm to 700 nm.

Photosynthetically Available Radiation (PAR) is defined as the total radiation available in this band. PAR is expressed either in µmol/m^2/s, i.e. the flux of photons per area per time, or as irradiance; W/m^2.

2.5 Light in the Arctic

At high altitudes, the rays from the sun strike the earth at a much more oblique angle than closer to the equator. The result is that the available light is spread out over a larger area, lowering the intensity. This lowering of intensity is proportional to the cosine of the angle of the rays to the surface, thus this is often referred to as the cosine effect.

Summer in the Arctic is characterized by midnight sun, and winter by the polar night. During the polar night, the sun never rises above the horizon, and during the summer the sun never sets. The midday, midsummer elevation of the sun is strongly dependent on latitude. It is 23.45° for the North Pole, 34° for Ny Ålesund and 43° for Tromsø. However, during the summer months the 24 h integrated irradiance can be almost as great in these latitudes as in the tropics, due to the sun being up all night (Sakshaug et al., 2009).

2.6 Attenuation of light

When radiation from the sun hits the atmosphere, it is attenuated by the gases it travels through. Oxygen, water and carbon dioxide all cause attenuation of the irradiance. At higher altitudes, because of the cosine effect, the light must travel a longer distance through the atmospheric gases. In fact, at latitudes $< 30°$, the distance is about twice as long as for zenith radiation, i.e. when the sun is directly overhead. Hence attenuation is stronger for light hitting the Arctic regions than for any lower latitude.

The attenuation at different wavelengths can also be seen from figure 2. At low solar altitudes, the direct radiation in the blue band is strongly attenuated, due to scattering in the atmosphere. Thus, direct radiation at low angles is mostly in the red band. As the angles approach zero, the available light will appear blue again, because direct radiation decreases much faster than indirect radiation. The indirect radiation in mostly in the blue band.

Another severe source of irradiance attenuation is clouds. Depending on the type of clouds and their thickness, the irradiance can be attenuated by as little as 20% and as much as 70%. During the Arctic summer, fog is often encountered over open water in calm weather. Also, the cloud cover associated with a passing low pressure system can cause the daily average surface irradiance to drop to 40%, compared to a clear day. In terms of irradiance, 40% corresponds to roughly 500 μmol/m^{2}s.

Clouds can also increase the radiation, if they are bright white and are not covering the sun. Light will then be reflected off them, and return to the earth surface. The increase in radiation can be as much as 10%.

Lastly, reflection is an important factor in the attenuation of light below the sea surface. Glassy water, i.e. water with no waves, reflects the most light, especially for low solar elevations. This sea state is normal in ice filled Arctic waters. A rougher sea state causes lower reflection.

The proportion of the downwelling irradiance that is reflected back into the atmosphere is known as the albedo. Albedo ranges from 0 to 1, 0 being total absorption, and 1 being total reflection. While the average albedo of the whole earth is 0.31, values for ice and snow is much higher, and can be as high as 0.97 for ice covered with fresh snow. The average albedo for the Arctic Ocean during summer is about 0.46.

The reflecting properties of ice causes a destabilizing, positive feedback in the climate change system. In cold periods, more ice forms, causing more light to be reflected and further lowering the ocean temperature. In the course of warm periods, the ice melts, which accelerates the heating of the ocean.

3. METHOD

The sensor used for this project is an *SPC2 Photon Imager*, developed by *Micro Photon Devices* [2]. The SPC2 Photon Imager is a camera, capable of imaging the light field by counting the number of incoming photons during the acquisition period. In the Marine Night campaign, the photon imager has been used to measure the light field in the Arctic polar night.

The photon imager is shown in figure 3.

[2] `http://www.micro-photon-devices.com/Contact`

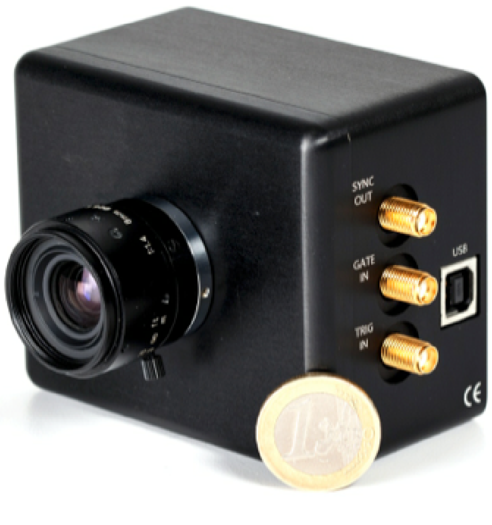


Fig. 3. The SPC2 Photon Imager

3.1 Capabilities of the photon imager

The information in this section is taken from the SPC2 User Manual.

The photon imager captures images of the light field in much the same way as a normal camera does. Light enters through the lens and is measured by the sensor on the imager's circuit board. The lens is mounted on a standard c-mount, which enables the camera to easily utilize many different lenses. The lens used for these measurements was a Tamron 8mm, with 1.4 maximum aperture. The field of view of the camera with this lens was estimated to be about 20°.

The light detecting sensor in the SPC2 is very different from the sensors found in conventional cameras. It consists of an array of 32×32 pixels. Each pixel is made up from a single-photon avalanche diode detector, with integrated electronics for digital processing. Because of this fully digital acquisition of the light signal, the camera has a high noise immunity, thus giving a very clean output signal.

The SPC2 is capable of acquiring images at up to 49 000 frames/s, with a negligible dead-time. Dead-time is the amount of time that passes between the end of one acquisition and the start of the next. The Application Programming Interface (API) also has a subroutine that enables dead-time correction, which can further reduce the effect of dead-time.

The SPC2 can be operated in two modes; *normal* and *advanced*. Each pixel contains an 8-bit counter, which counts the number of photons detected during the exposure time. In normal mode, the exposure time is fixed to 20.74 μs, ensuring that the counter will not overflow. Longer exposures are possible by integrating multiple frames. Because of the noise immunity, this integration does not increase the noise. In advanced mode, the user can set the exposure time, for better control of the camera. Longer exposures makes counter overflow possible, and this must be checked for and handled by the user.

Other parameters include number of acquired frames, and number of frames to integrate on each acquisition[3]. When the camera is commanded to acquire images, it returns a set of images. The number of images in this set is the number of acquired frames, which must be set beforehand. In normal mode, the number of integrated frames is set to decide the exposure time.

3.2 Application Programming Interface (API)

The SPC2 Photon Imager includes an API, written in the `C` language. An API is an interface between two systems, in this case the SPC2 camera and the user's application. It is used for writing programs to control the camera, and includes multiple functions for connecting, setting parameters and acquiring images, to name a few. The API was used for writing the custom software that was used on the campaign. A few of the most important functions are detailed below.

`SPC2_Constr` is used for getting a handle to the camera. A handle is an object which is a unique identifier to the camera, and it is used to communicate with it. The handle must be acquired before any other functions can be called.

`SPC2_Set_Camera_Par` sets the camera parameters; exposure[4], the number of frames per acquisition, and the number of integrated frames. This function must be called before acquiring any images.

`SPC2_Snap` acquires a set of images, as specified in the parameters. The acquired images are stored in the working memory of the camera, and must be retrieved by one of the other functions; `SPC2_Average_Img` or `SPC2_Save_Img_Disk`.

`SPC2_Average_Img` returns one image, which is an average of the previously acquired images.

3.3 Software developed for the project

A software package (from now on referred to as the *program*) was developed during the campaign, to enable the SPC2 to measure the ambient light in the polar night. Some interesting light events were expected on 18 and 19 January. To ensure that the sensor would be operational in time for these events, a minimal set of specifications were developed by the end users[5]. The sensor had to be able to:

- Measure the ambient light once per minute
- Store the measurements safely

The program acquires an image approximately once every minute. Each returned image is the average of 100 images, each of which is the integrated result of 5 frames. This average is then stored in a database. In addition, the program calculates the photon count per second, and stores that in the same database. The photon count is calculated by summing the values of each pixel in the image, which represents the photon count detected by the representative pixel on the sensor. This sum is then divided by the acquisition time, to get the photon count per second.

[3] In normal mode

[4] Only in advanced mode.

[5] The users in this context are Prof. Geir Johnsen, Dep. of Biology, NTNU, and Prof. Jørgen Berge, Dep. of Arctic and Marine Biology, University of Tromsø.

3.4 Database

The database system used in the program is a *SQLite* database. From the web page[6]:

> SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine. SQLite is the most widely deployed SQL database engine in the world. The source code for SQLite is in the public domain.

The above description highlights why SQLite was chosen for this project. It is self-contained, meaning that the program can run on any machine without having to set up a database server. That the source code is in the public domain means that anyone is free to use the software for any purpose, i.e. no licence is needed[7]. Finally, SQLite is used in many large projects by well-known companies[8], which implies that it is a well tested and stable software product.

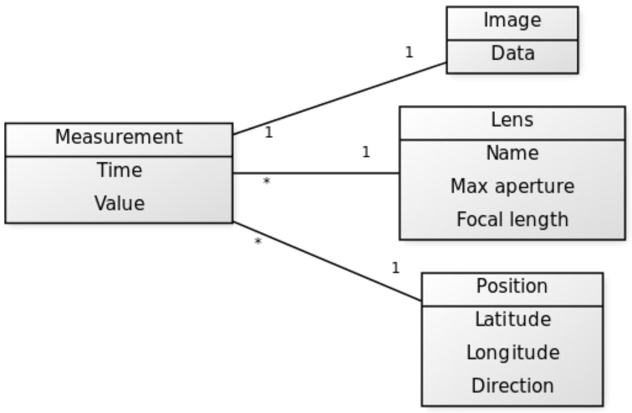


Fig. 4. The database schema for the program

Figure 4 shows the database schema that was used for the program. The database consists of four tables:

- *Measurement* is the central table. It stores the time and value of each measurement. The time is local time; the value is in photons/s. This table has a many-to-one relationship to both *Position* and *Lens*.
- *Position* records the position where the measurement was taken. Latitude and Longitude are the coordinates, and Direction is the direction the camera was facing when the measurements were taken.
- The table *Lens* stores information about the lens used for the measurements. It is identified by its name, maximum aperture and focal length.
- *Image* contains the image returned from the camera. It is stored as binary data, and can be retrieved in a `double` array of length 1024. *Image* has a one-to-one relationship to *Measurement*.

[6] `http://www.sqlite.org`

[7] `http://www.sqlite.org/copyright.html`

[8] `http://www.sqlite.org/famous.html`

3.5 *Experimental setup*

The SPC2 camera was set up outside the Marine Lab in Ny-Ålesund. The Marine Lab is situated just outside the settlement and next to the pier. It has a veranda facing northwards, towards Kings Bay. This is where the camera was deployed, with the lens facing out towards the bay.

When doing light measurements, light pollution is a problem. Light pollution is measured light from artificial sources, such as street lights or headlamps on cars or boats. To get reliable measurements, light pollution must be reduced as much as possible. At the Marine Lab, this was accomplished by covering all the windows with black plastic. The street lights around the lab had also been switched off for the duration of the campaign. Some light pollution from cars, and especially the research vessel *R/V Helmer Hanssen*, was unavoidable. The times of these events were noted, so they could be disregarded in further study of the data material. In addition to these precautions, the camera was positioned against the north wall of the Marine Lab, facing towards the bay. This way, the Marine Lab shielded the camera from any artificial light coming from the settlement of Ny-Ålesund.

The camera was mounted on a tripod, with its power and data communication cable leading from the camera through an open window, to a computer. The computer was running the software described in section 3.3. At this point, the camera needed no attention from an operator. It was left to take measurements from 18 – 26 January, 2014. In addition to the camera, another prototype irradiance sensor was running. This sensor is described in section 3.6.

The camera is not built for outdoor use. Thus, it needed to be protected from the weather. Protection was fashioned out of plastic bags. It was rudimentary, but sufficient for the weather conditions during the campaign.

3.6 *IMO irradiance sensor*

The IMO irradiance sensor is developed by *In-situ Marine Optics* [9] located in Perth, Australia. It is a prototype, with model name *ISSU* and serial no. 2. It has been developed to address the problem of measuring very low levels of irradiance, such as is seen during the Arctic polar night.

The IMO sensor measured photons as digital counts, from which E can be calculated. It also measured temperature in °C.

In contrast to the SPC2 camera, the IMO sensor was purpose built for marine use. It is built for underwater use, and is equipped with a battery pack and a logging computer. Thus, it is fully autonomous and able to take measurements in a much wider range of conditions.

3.7 *Limitations*

The field of view of the camera was estimated to be 20°. Thus, the camera will only capture part of the lightfield. For most of the time, the measurements were done in diffused light. Diffused light is the opposite of direct light, meaning that it is reflected or transmitted through a medium, such as clouds. This is the available light on an overcast day. In these conditions, it was assumed that the camera captured a representative value of the total irradiance, because the irradiance would be roughly equal in every direction. On days with a point source of light, such as the moon, the measured values would be drastically different depending on the orientation of the camera. A camera pointing at the moon would measure much higher values than if it was pointing elsewhere.

The camera is sensitive to temperature changes. The camera was tested indoors, in close to constant ambient light. During this time, the temperature increased noticeably. Also, the photon count kept increasing, although the ambient light was more or less constant. During this test, the sensor was programmed to capture images every second. During the measurements, the imager was placed outside, in temperatures between $-16\,°C$ to $3\,°C$. Thus, heating of the camera was not a problem. If the camera is to be used in warmer temperatures, especially indoors, the temperature issue would have to be addressed.

The camera is not made for underwater use. An underwater housing must include battery power and a computer running the necessary software, making it completely autonomous. For now, measurements can only be made in air.

The software for the camera is of very limited capability. The only thing it does at the moment, is to capture images every minute, and save the average image and the sum in a database. Full featured software should be able to capture and display test images, set parameters and configure different types of logging.

3.8 *Calibration*

The values from the IMO sensor was used to calculate a rough calibration constant for the SPC2 camera. By matching the peaks in the data set from the IMO sensor with the same peak from the camera, and assuming that the data from the camera was representative for the entire light field [10] , a rough calibration constant was calculated.

The data from the IMO sensor was filtered using a Butterworth filter. This reduced most of the dark current, so the data was easier to analyze. Disregarding the outliers, which are likely anthropogenic in nature, the maximum of the dataset was 1.8232×10^{-4} This occurred on 21 January, 12:26:47 CET. The corresponding time was found on the graph of the data from the camera. The photon count at that time was 1.9298×10^{7}. Thus, the calibration factor was:

$$\frac{1.8232 \times 10^{-4}}{1.9298 \times 10^{7}} = 9.4476 \times 10^{-12} \tag{1}$$

Using this factor, the values from the camera were converted from photons/s to µmol photons/m^{2}s.

4. EXPERIMENTAL RESULTS

4.1 *SPC2 photon imager*

The measurements done with the SPC2 camera is shown in figure 5. The horizontal axis shows the time; each tick

[9] `http://www.insitumarineoptics.com/contact.html`

[10] See discussion in section 3.7

mark is placed at noon on the given date. The vertical axis shows E_{PAR} values; the combined value of E for all wavelengths in the 400 nm to 700 nm band.

On most days, there is a peak in the measurements around noon. The peak is lower or higher, depending on the weather conditions. Some days, like 20 January, hardly has a peak at all. This was a snowy day, with a low cloud cover.

Micro Photon Devices claim that the sensor has a very high noise immunity, as described in section 3.1. This is verified by figure 5. There is little, if any, trace of electronic noise.

Figure 5 shows the values in E_{PAR}. The output from the program is in photons/s. A rough calibration has been done, based on the values from the IMO sensor. This calibration procedure is explained in section 3.8.

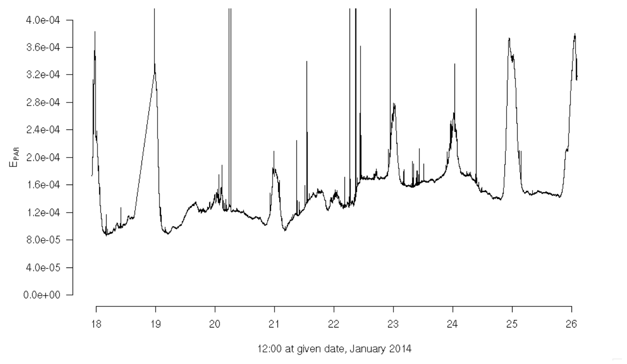


Fig. 5. Light conditions in Ny-Ålesund, measured with the SPC2 camera.

On clear days, such as 19 January, there is a clear peak at noon. This corresponds well with photographs that were taken at the same time. This is shown in figure 6. The background photograph is taken with a fisheye lens just outside the Marine Lab. The blue line on the plot shows exactly when the photo was taken.

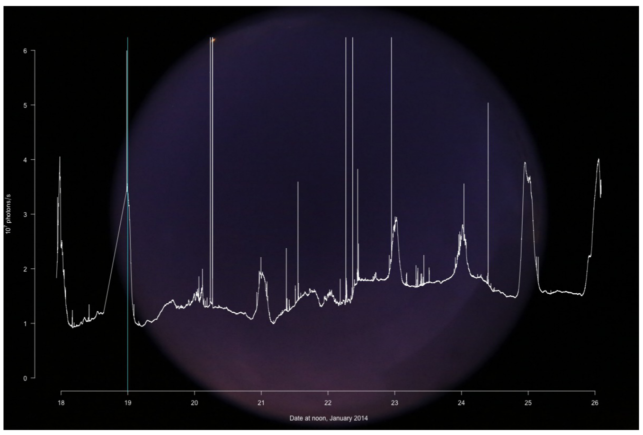


Fig. 6. Light conditions at midday. Background photo: Geir Johnsen.

The graph also corresponds well with the conditions at night. Figure 7 shows the irradiance values at night, under a clear sky. The low light levels can be verified both on the photograph and on the plot.

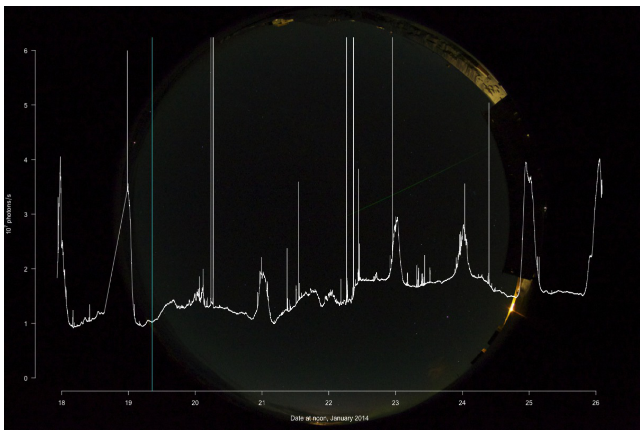


Fig. 7. Light conditions on a clear night. Background photo: Geir Johnsen.

The plot also shows multiple isolated spikes. The spikes are several orders of magnitude larger than the average measurements. These spikes most likely correspond with artificial light sources, such as headlamps on vehicles, and the floodlights of the research vessel *R/V Helmer Hanssen*, which docked at the pier on several occasions during the campaign.

There is some missing data roughly between 03:00 and noon on 19 January. This was due to a computer restart following system updates. It was discovered in the morning and the program was restarted. The missing data is seen as a straight line in the plot between tick marks 18 and 19.

4.2 IMO sensor

The irradiance measurements from the IMO sensor are shown in figure 8. These measurements are clearly more noisy than the corresponding values from the SPC2 camera. However, there are clear peaks around noon for both Tuesday 21 January and Wednesday 22 January. This corresponds well with the measurements from the camera; both showing a lower peak on the 22nd.

The IMO sensor came with calibration instructions to convert the raw measurements into E_{PAR}, which is shown on the plot.

5. FURTHER WORK

The SPC2 camera as an irradiance sensor is a prototype. In its current form it lacks many features. Detailed in this section are suggestions for further work that would make the camera a functional sensor for measuring irradiance in the marine environment.

5.1 Calibration

The conversion from photon count to irradiance is a rough estimate. For more refined results a conversion factor for the sensor must be calculated. Finding the conversion factor would most likely be achieved by measuring different light sources with known irradiance values, and calculating the conversion factor based on those measurements.

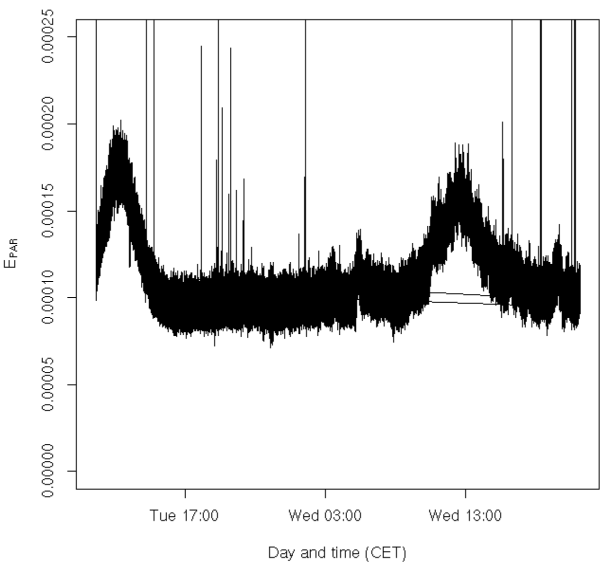


Fig. 8. Light conditions in Ny-Ålesund, measured with the IMO sensor. Measurements taken between Tuesday 21 January and Thursday 23 January.

5.2 Software

The software has only basic functionality. In a full-featured system, it should be able to

- Set the camera parameters
- Take test images and display them to the user
- Adjust the parameters online, according to preset goals
- Show a live update of the acquired data
- Run in both headless mode, for autonomous logging, and with a graphical user interface

5.3 Autonomous unit

The prototype is limited in its usefulness by the fact that it needs to be connected to a computer to run. A functional system should be able to operate autonomously. Thus, it would need both a computer to run the software, and a power supply. For the computer, it should be as small as possible, and preferably running a low resource demanding operating system. That would imply porting the program for the camera to the new operating system also. The power demand of the whole system must then be calculated, to come up with a power supply that is reasonably small and lightweight, but still can run the camera for an acceptable length of time.

6. CONCLUSION

This paper has described the development of a prototype irradiance sensor, using the SPC2 Photon Imager, which measures the photon count of the ambient light field. Development of the sensor included both development of the necessary software, deployment of the sensor in the Arctic environment during the polar night, and a comparison of the measurements from this sensor with those of another prototype.

The results from the prototype sensor were promising. The measurements clearly showed differences in the ambient light levels throughout the day. These measurements will be of value to researchers studying life in the high Arctic during the polar night.

REFERENCES

Berge, J., Cottier, F., Last, K.S., Varpe, O., Leu, E., Soreide, J., Eiane, K., Falk-Petersen, S., Willis, K., Nygard, H., Vogedes, D., Griffiths, C., Johnsen, G., Lorentzen, D., and Brierley, A.S. (2009). Diel vertical migration of Arctic zooplankton during the polar night. *Biology Letters*, 5(1), 69–72.

Sakshaug, E., Johnsen, G., and Kovacs, K.M. (2009). *Ecosystem Barents Sea.* Tapir Academic Press, Trondheim.

Young, H.D. and Freedman, R.A. (2012). *University Physics.* With Modern Physics Volume 1, . CHS. 1-20. Pearson Education.

Testing and Verification of a Low Light Irradiance Sensor

Lars Lønne [*]

[*] *Norwegian University of Science and Technology, Trondheim, Norway (e-mail: larslonn@stud.ntnu.no).*

Abstract: A new sensor has been designed and developed for measuring the low levels of irradiance in the Arctic polar night. As a part of the development, a number of experiments have been carried out to ensure that the sensor performs as expected. The experiments test the sensor's ability to capture the dynamics of the changing light and its measurement noise. The sensor's correctness is also tested, by comparing the output to another sensor. The tests are mostly positive, indicating that the sensor works as expected.

1. INTRODUCTION

Researchers working at the University Center on Svalbard (UNIS) have discovered that life forms in the Arctic might be dependent on the light in the polar night (Berge et al., 2009). The level of illumination during the polar night is often too low for the human eye to register. Also, commercially available light meters are not sensitive enough to measure these microscopic light values. The need to measure these low light levels has led to the testing and development of a new irradiance sensor, which is presented in this paper.

The new sensor is based on a product from Micro Photon Devices (MPD) [1], called the *Single Photon Counter Camera (SPC2)*. The SPC2 is capable of single photon detection, with low dark counting rates and capturing frame rates up to 48 000 kHz. The camera was tested during the January 2014 campaign of the *Marine Night* research project in Ny-Ålesund, Svalbard, and showed promising results. Most importantly, the camera was able to capture the light dynamics of the polar night, with very little measurement noise. See Lønne (2014) for details.

Together with the camera, the new sensor also consists of two pieces of newly developed software. The sensor is built around the client/server model. A server application runs on a computer that is connected to the camera, and the user interacts with the sensor through a client application, which runs in a web browser. The server is responsible for communicating with the camera, saving measurements from the camera in a robust and safe way, and enabling clients to take measurements and retrieve data. The client is responsible for presenting the user with an interface that offers sufficient control, but at the same time is easy to use. The overall structure of the new sensor system is shown in figure 1.

Testing and verifying the sensor is part of the overall system testing. Before the sensor can be used in the field, users must have confidence in the equipment, and be relatively sure that it will perform as expected.

[1] Micro Photon Devices, Via Stradivari 4, I-39100 Bolzano, Italy

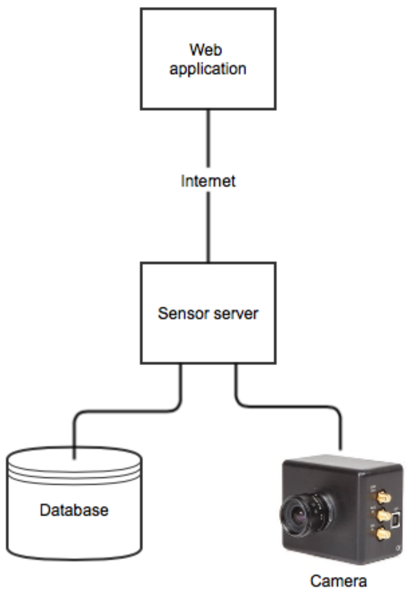


Fig. 1. The structure of the sensor system.

This paper describes the initial testing and verification of the sensor. Simple test cases are performed, to verify that the sensor performs as expected, and that it registers reasonable irradiance values. Section 2 lists the expected functionality of the sensor, and what the end users expect it to be capable of. Section 3 presents some theory on the different ways to measure illuminance and irradiance, and presents the foot-candle, which is used in one of the experiments. Section 5 describes the setup of the different tests, and briefly discusses their limitations and their strong points. Section 4 discusses the different calibration options that the sensor offers, and why they may be useful. Section 6 presents the results of the tests. Section 7 presents the conclusion of the tests, and finally, section 8 lists possibilities for how the sensor's capabilities can be extended in the future.

2. EXPECTATIONS

The SPC2 was initially tested in January 2014, during the polar night in Ny-Ålesund, Svalbard. Ny-Ålesund is located at 78°55′30″ N, 11°50′20″ E and is well within the polar night region. The initial tests were successful, in the sense that the camera was able to capture the little light that was present at this time of the year with high fidelity. Also, the results were comparable with those measured by other sensors, taken at the same location.

The new sensor is expected to perform as well as last time. It is also expected to record time series of the light conditions in Ny-Ålesund for an extended period of time. Thus, the software is expected to be reasonably robust, and the storage facilities must be able to safely record the data. The data must be protected against such eventualities as power outages and hardware failure.

Together with the users of the sensor, a specification of the sensor functionality was drawn up before development began. The specifications are discussed in detail in the résumé that accompanies this paper. One point that is contrasting to the last iteration of the sensor, is that it must have remote control capabilities. The users expect to monitor and control the sensor from any other location, over the internet.

Ultimately, the most important expectation is that the sensor stores correct measurements, and that the users are confident in its continued, safe operation.

3. THEORY

3.1 PAR measurements

The sensor measures light as a count of the incoming photons in a given time frame. The count is then converted to µmol/m^{2}s, which is a standard way to quantify Photosynthetically Active Radiation, or PAR. All photons in the PAR band carry enough energy to start photosynthesis. It is the number of photons that count, not the total energy. Thus, it is convenient to measure irradiance on the quantum scale when talking about photosynthesis (Sakshaug et al., 2009).

PAR can also be expressed in energy units; W/m^2. This is preferred when doing energy calculations, such as measuring solar energy flux in the polar regions.

Converting between µmol/m^{2}s and W/m^2 is difficult unless the spectral distribution is known. Photons carry different amounts of energy for different wavelengths, and this must be accounted for in the conversion.

3.2 The lux scale

Another scale for measuring illuminance is the *lux scale.* This scale was designed for measuring illuminance indoors, and for architecture. The scale is therefore adjusted for the spectral sensitivity of the human eye. The human eye responds well to light in the green band, while plants absorb light mostly in the red and blue bands of the spectrum. In fact, algae and higher plants hardly respond to green light at all.

In Sakshaug et al. (2009), there is an "average" conversion factor from klx to µmol/m^{2}s in the range 14–18. This conversion factor must be used with care, as the spectral composition can change a lot, especially in water. For sunlight however, the spectral distribution varies little as long as the elevation of the sun is greater than 10°.

3.3 The foot-candle

The foot-candle, abbreviated fc, is a unit of illuminance. It is the amount of light one foot away from a point source of one candle. It is equal to one lumen per square foot.

The fc unit is intuitive, and makes possible a rough procedure to calibrate a light meter. One foot candle is roughly equal to 10 lux, and 1 klx is roughly equal to 14 µmol/m^{2}s to 18 µmol/m^{2}s (see Sakshaug et al. (2009), appendix A).

4. CALIBRATION PROCEDURES

There are three elements of the camera that must be calibrated; the sensor's active area diameter, noise and the photon detection efficiency. The information on calibration comes from communications with the manufacturers of the camera, Micro Photon Devices.

Active area The active area of the sensor is perfectly determined by the manufacturing process. It needs no further calibration. The active area diameter is 20 µm.

Noise Noise is called *Dark Counting Rate (DCR)* by MPD. It is the photon count registered by the camera in a completely dark environment. The procedure to calculate it is as follows:

(1) Put the camera in a completely dark environment, and acquire a frame with a long exposure time. MPD recommended 10 s.
(2) Divide the count in each pixel with the exposure time to get the DCR for each pixel. Use the data as-is, or sum it to get DCR for the whole pixel array.
(3) Check that the measured DCR is compatible with MPD's measurements, which are provided with the camera.

Photon detection efficiency Measuring the photon detection efficiency is a simple concept, but difficult in practice. The difficulties lie in that the procedure calls for calibrated monochromatic source, and a method for determining the exact power the camera sees at the given wavelength.

MPD has performed tests of photon detection efficiency for every wavelength from 400 nm to 1000 nm, with a 50 nm step. The results are included with the camera, and are also shown in figure 2. The dotted lines show the PAR band, i.e. 400 nm to 700 nm. The camera counts photons up to 1000 nm.

If the spectral distribution of the measured light is known, figure 2 can be used directly to calibrate the sensor. The raw measurements would be saved by the sensor, and

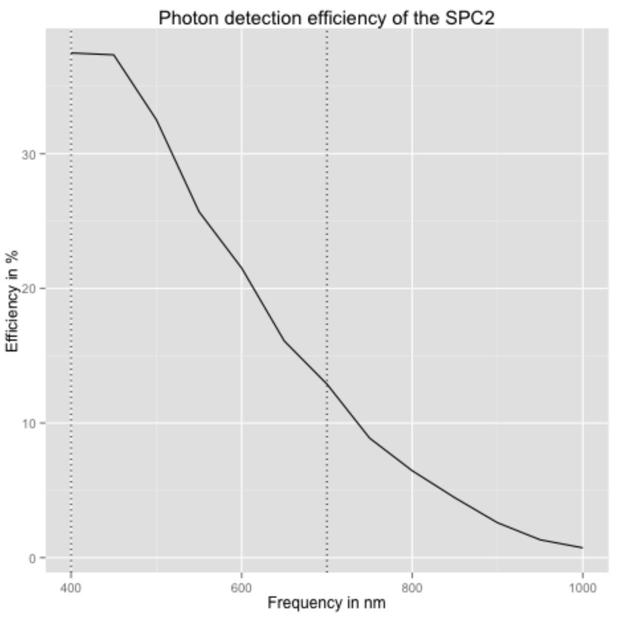


Fig. 2. Photon detection efficiency.

adjusted later using a weighted average, with the weights taken from the spectral distribution of the incoming light.

A more realistic scenario is that the spectral distribution is unknown. In the case of the polar night, all available light is indirect. The indirect light is mostly blue (Sakshaug et al., 2009). The SPC2 is also most sensitive in the blue band, making the polar night an ideal environment for this sensor. The users will have to make some assumptions about the actual spectral distribution, and calibrate the measurements according to these assumptions.

5. TEST SETUP

5.1 24 hour time series

This test aimed to capture the light dynamics in a 24 hour period. The test was performed in Trondheim, Norway. The camera was situated in a window, inclined at roughly 30° from the horizontal, with the lens pointing to the sky. The test was performed in an urban area, so some light pollution should be expected, but the camera was positioned to avoid pollution as much as possible.

To protect the camera's buffers from numerical overflow, the camera operates in *normal* mode. In this mode, the camera captures frames with constant exposure time of 20.74 µs. Multiple frames are integrated, or summed, to achieve a longer exposure time. For this experiment, the exposure time was set to 10 ms, which corresponds to 482 frames per image. 482 frames does not add up to exactly 10 ms, but rather 9.997 ms, because the exposure time can only be set in discrete steps.

No calibration was done prior to this experiment, so the calibration factor was set to 1. Each data point is the average of 5 images.

5.2 Dark counting rate

When the camera is placed in a completely dark environment, it will get counts from the photon counters. These counts are false positives, and contribute to the noise in the measurements. The noise is referred to as the Dark Counting Rate (DCR).

To measure the DCR, it was necessary to capture images in a completely dark environment. This was achieved by capturing images in a dark room. To further minimize stray light, the lens was removed from the camera, and the sensor was covered with a cap. The camera was then positioned face down on a table.

Measuring the dark counting rate requires that the images are acquired with a long exposure time. Because the camera operates in normal mode, the exposure time has to be a multiple of 20.74 µs, and the number of captured frames is limited to 65535. Thus, the maximum exposure time is limited to ≈ 1 s. The exposure time was set to 48216 frames, which is ≈ 1 s.

Multiple images were captured with these settings, and formed the basis for the DCR calculations.

5.3 Measuring the irradiance of a candle.

The camera was positioned 1 foot away from a burning candle, facing the light. The camera and the candle were placed in a dark room. Multiple irradiance measurements were taken of the candle's flame. The measurements were then converted to the unit foot-candle, to verify that the sensor's output was reasonable.

This experiment was performed as a rough verification of the sensor's output values. In Sakshaug et al. (2009) a crude conversion between kilolux and µmol is given as

$$1\,\mathrm{klx} \approx 14 - 18\ \mu\mathrm{mol/m^2 s}$$

Since 1 foot-candle is approximately 10 lx, a candle can be used as a basic verification of the sensor's output.

5.4 Comparing the SPC2 to the Walz Diving PAM

To investigate if the SPC2 were measuring reasonable values, it was compared to another light meter. The comparison was done with a Diving PAM underwater fluorometer manufactured by Walz [2]. The Diving PAM is

> *[...] a worldwide unique instrument for studying in situ photosynthesis of underwater plants, including sea grasses, macroalgae, and zooxanthellae in corals.* [3]

The Diving PAM has a lower limit of reliable measurements at about 1 µmol/m^{2}s.

The experiment was performed on 19 December 2014, in the afternoon between 14.30 and 15.00 local time. The two sensors were set to measure the outside ambient light. The weather was clear.

The light sensor on the Diving PAM is much smaller than the lens on the SPC2. The SPC2 is believed to be able

[2] Heinz Walz GMBH, Eichenring 6 - 91090 Effeltrich, Germany
[3] From the company's website: www.walz.com

to capture a larger light field than the Diving PAM. Also, the light sensor on the Diving PAM is cosine corrected, meaning that a diffuser is placed over the light sensor to account for light coming in at different angles. These two differences will probably account for some deviations in the results.

5.5 Capturing the light field as an image

Lastly, the sensor was tested as a more conventional camera, to see how well it performed when capturing images of the light field. This experiment used the *Capture test image* function of the web application, which captures an image of the light field, converts it to the PNG format and displays it to the user.

Two images were captured, in normal daylight conditions indoors. In the first image, only the ambient light was present. The second image was captured with the same settings, but with an artificial light source pointed at the camera, to see if it is possible to discern light sources on the images.

6. RESULTS

6.1 24 hour time series

Figure 3 shows the result of measuring the irradiance in Trondheim for 24 hours, on 15 and 16 December 2014. The graph shows a couple of interesting facts.

First, irradiance values when the sun is below the horizon is approximately constant. The values are in the range 0 µmol/m^{2}s to 0.5 µmol/m^{2}s. Typical irradiance values at night is between 0 and 1 µmol/m^{2}s, which proves that the values from the sensor are perfectly reasonable. The sky was clear on the night of the 15th, which corresponds well with the constant measurements. Some small disturbances can be seen; they are believed to be caused by light pollution from the urban environment.

On 16 December, the sun rose at 09:58, local time. A detail of figure 3 is shown in figure 4, which shows the period just before and after sunrise. The graph is seen to rise sharply just after 08:30, which agrees well with the timing of the sunrise.

As the sun comes over the horizon, the sensor seems to have saturated. This was confirmed by checking the original measurement data. All counters had saturated.

Figure 5 shows a detail for the sunset. It is almost a mirror image of the sunrise. The local sunset was at 14:32, and the irradiance can be seen to sharply drop some time after that. The saturation of the sensor hides the dynamics between 14:30 and 15:00 from view. After 15:00, the irradiance is seen to drop steady to a constant value below 1, as expected.

6.2 Dark counting rate

The camera's sensors are laid out in an array with dimensions 32 × 32. A visualization of the dark counting rate for the entire array is shown in figure 6. Dark colors correspond to low dark counting rates, and light colors,

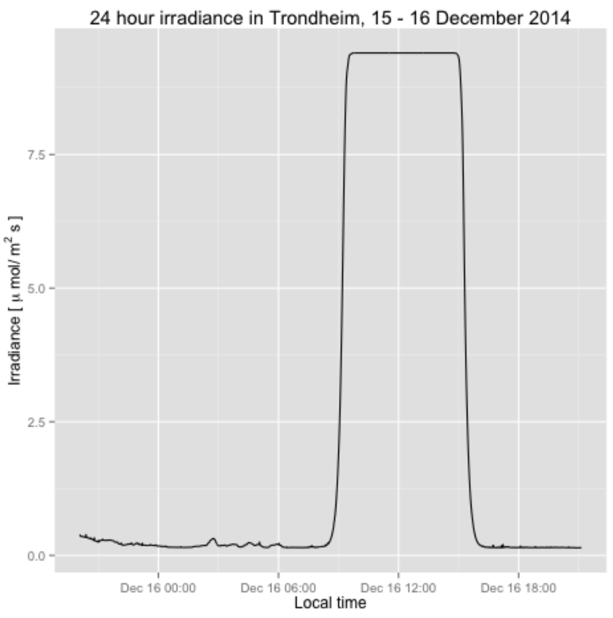


Fig. 3. 24 hour time series

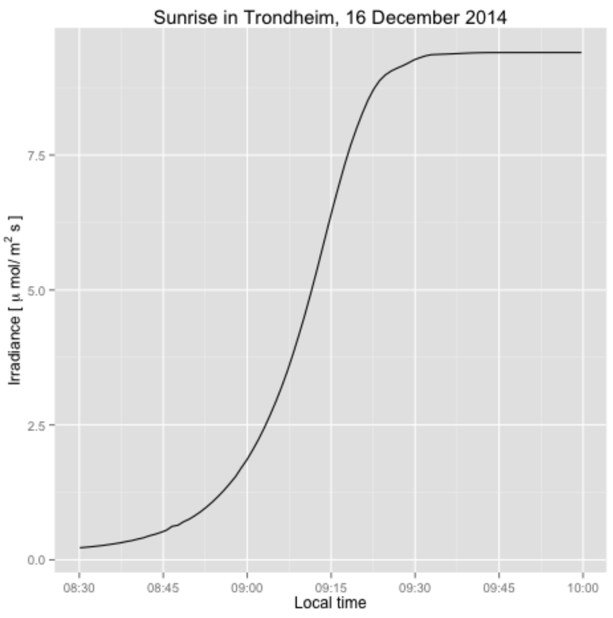


Fig. 4. Sunrise detail for the 24 hour time series

toward yellow, signify high dark counting rates. Because of the long exposure time, some of the pixels have saturated.

MPD guarantees that the dark counting rate will be within certain bounds. Specifically, they guarantee that half of the pixels will have a dark counting rate below 4000 cps (counts per second). Also, 75% of the array should have dark counting rates below 25000 cps.

For the test image shown in figure 6, 729 pixels have a DCR value under 4000 cps, and 841 pixels have a DCR value less than 25000 cps. In percentages, that is 71.19% and 82.13%, respectively.

The lowest dark counting rate in the array was 1955 cps. The maximum is unknown, because the pixels with the

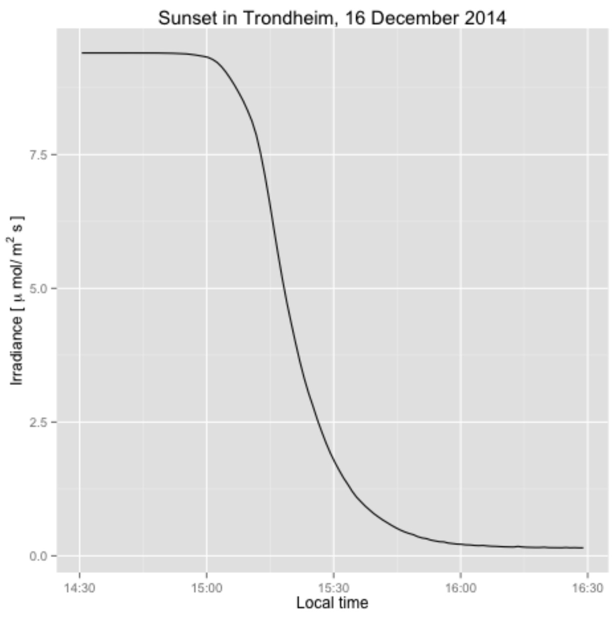


Fig. 5. Sunset detail for the 24 hour time series

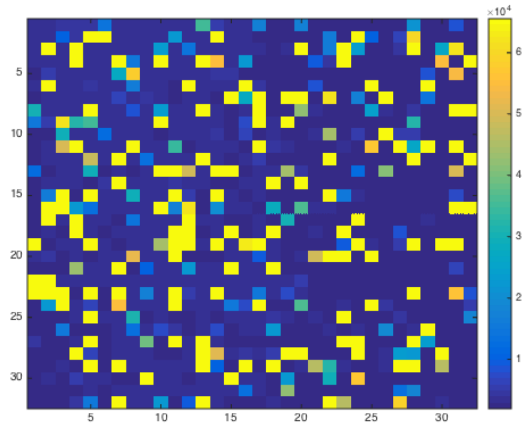


Fig. 6. The Dark Counting Rate (DCR), or noise, for all the pixels in the camera.

highest counts had saturated. The mean of the DCR values was 13815, and the standard deviation was 22039. Figure 7 shows the dark counting rates in a histogram, where the y axis represents the number of pixels, and the x axis represents the counts per second. Clearly, most of the pixels have DCR values which are less than 10000 cps.

6.3 *Irradiance of a candle*

Figure 8 shows 10 irradiance measurements of a burning candle at a distance of approximately 1 ft. The red line shows the mean value, which is 8.1283 µmol/m^{2}s.

1 foot-candle is approximately 10 lx. 1 klx is approximately 14–18 µmol/m^{2}s, see section 5.3. Converting the measured irradiance of the candlelight to foot-candles, gives a value which is about 50 times too large, before taking into account the photon detection efficiency.

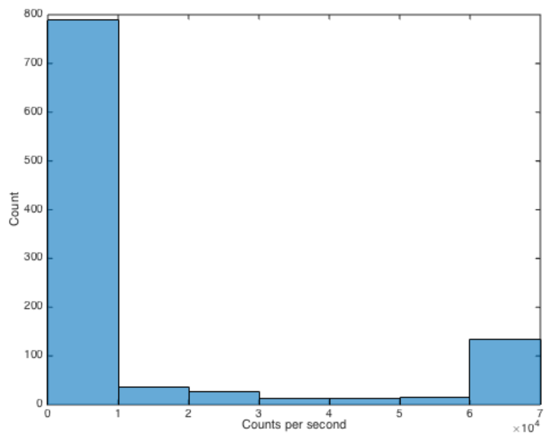


Fig. 7. Histogram showing the dark counting rates for all the pixels in the camera.

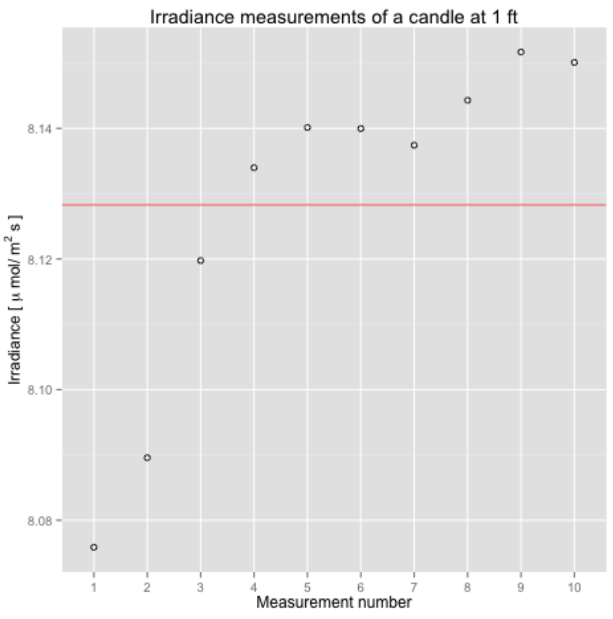


Fig. 8. Irradiance measurements of a candle. The red line shows the mean of the dataset. Note that the y axis does not start at zero.

There are multiple possible sources for this error. First, the experiment is very simple. It is not known for certain how much light is emitted by the candle; it was only assumed to be 1 foot-candle at a distance of 1 ft.

The conversion factor in Sakshaug et al. (2009) is valid in sunlight. In this experiment, it has been assumed that the conversion factor holds, which it might not. Converting between µmol/m^{2}sand lx is dependent on the spectral distribution, and without knowing it, the conversion is suspect.

Other factors that might have affected the result, but unlikely in a major way, are reflectance of the light from nearby walls, and background light.

Finally, the sensor might be in error. However, other tests, such as the 24 hour irradiance measurements, have proved to be reasonable. Thus, it seems prudent to rule out the possible sources of error before assuming that the measurements are in the wrong.

Clearly, other experiments must be performed to assess the correctness of the sensor. This is a task for future studies.

6.4 Comparison with the Walz Diving PAM

The Diving PAM measured values in the range 3 µmol/m^{2}s to 6 µmol/m^{2}s, while the SPC2 measured 8 µmol/m^{2}s to 9 µmol/m^{2}s. Thus, the results differed with a factor of about 2.

The Diving PAM has a much lower sensitivity, which probably accounts for some of the error. Also, the SPC2 has never been calibrated against a known light source, meaning that its absolute values must be treated with some suspicion.

The values were within the same order of magnitude, which implies that the SPC2's measurements are within reasonable limits. Further calibration and a more rigorous procedure is needed to confirm this more accurately.

6.5 Images of the light field

An image of the light field indoors is shown in figure 9. The image was captured in a room with windows in the morning, when the sun was up. The exposure time is 15 ms.

The resolution of the image is 32 × 32 pixels, where each pixel's value is based on the photon count of a single photon detector. It is nearly impossible to recognize anything in the image, although contrasts show up reasonably clearly. An object with a high contrast to its background should be possible to recognize.

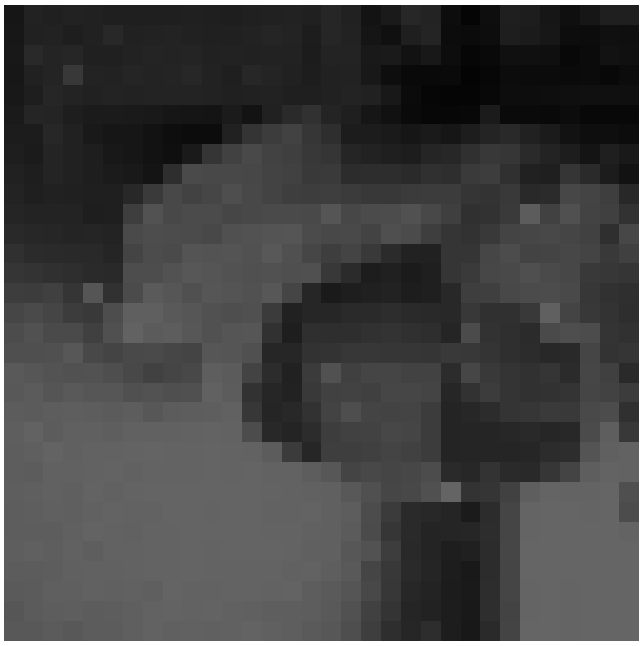


Fig. 9. Image of the light field.

Figure 10 shows an image of the same room, with a point source of light pointed towards the camera. The point source is a flashlight. The camera is not saturated, as the image is still gray; saturated pixels are completely white.

The contrasts from figure 9 is gone, presumably because of the new light source. It is nearly impossible to discern anything in the image, except for some darker patches in the corners.

Most importantly, it is hard to identify the light source in the image. In figure 10, the light source is pointed out with the arrow. The light from the flashlight causes increased counts in almost all pixels, causing the entire image to become more indistinct.

When using the sensor to capture images of the night sky, or the polar night environment, users have expressed that it would be advantageous to be able to identify point light sources, such as the moon. On the basis of these tests, it seems like it would not be possible to identify point sources from the images. The identification would have to be done some other way, e.g. by simultaneously taking images with a traditional camera.

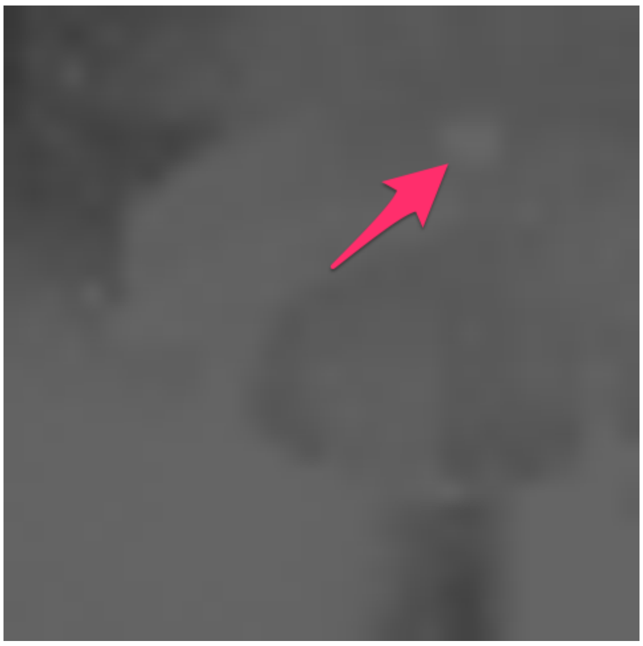


Fig. 10. Image of a point source of light. Light source marked with an arrow.

7. CONCLUSION

The experiments that have been described in this paper, were meant to test the new sensor system, and confirm that the measurements it provides are reasonable. Most of the experiments were successful in this regard.

The 24 hour time series demonstrated that the sensor is able to measure the dynamics of the light field. It also established that the sensor has a saturation limit which can cause measurement problems when the difference in irradiance is large. For measuring light in the polar night, this should not pose much of a problem, since the irradiance levels are relatively low throughout the day. For other uses, this issue will have to be addressed, either by manual intervention or by extending the sensor to set the exposure time autonomously.

The dark counting rate experiment shows that the DCR values are very reasonable, and well within the guaranteed levels from the manufacturer. Additionally, the experiment increases the confidence in the new sensor system, in that it captures and delivers the correct images.

Measuring the irradiance from a candle was not successful. No conclusions will be drawn from this experiment, because there are too many uncertainties, which are discussed in section 6.3.

Comparison with the Walz Diving PAM showed that the sensor's output falls in a reasonable range. This indicates that the sensor's measurements needs further calibration, but that it is absolutely fit for the purpose.

Finally, the images of the light field goes some way in demonstrating the camera's effectiveness in visually describing the incoming light. With some training, it might be possible to discern points of interest on these images. The resolution is so low, however, that all such identifications must be treated as uncertain.

8. FURTHER WORK

8.1 Automatic adjustment of exposure time

As was seen in the 24 hour time series experiment, there is a need for adjusting the exposure time when the light levels change. In the interest of minimal manual adjustment, it would be preferable if the sensor adjusted the exposure time automatically, to avoid saturating the photon counters when light levels rise above a critical level.

Each element in the captured image is a 16 bit counter, meaning that the maximum value it can represent is 65535. The automatic adjuster algorithm could be implemented as a regulator, with the aim of keeping the mean value of the measurements well below this limit, but also above the dark counting rate. By capturing a test image, the mean of the counters can be calculated, and the exposure time set accordingly.

8.2 Calibration against a known light source

For the sensor to be able to reliably measure irradiance in absolute values, it needs to be calibrated against a known source of light. The spectral signature of the light source must also be known, such that the calibration can take into account the photon detection efficiency at different wavelengths.

8.3 Alarms

The specification of the sensor lists alarms as a priority B task. This is not yet implemented due to time constraints.

The sensor should alert the user of any trouble it encounters during operation. Examples of problems could be hard drive failure, causing one of the databases to shut down. It could also be other sources of hardware failure, such as problems connecting to the camera, or problems with the power supply. The alarm system should also warn the user in the event of software failures, such as failure to save measurements. A more sophisticated alarm system could also monitor the output of the sensor, and issue warnings when the output seems suspect. Problematic output could be very high or low measurements, or a significant increase in noise, which could be an indication of hardware problems.

The alarms would have to be issued to the user through the user interface, e.g. on a list on the main page of the web application. The interface would need some way of dismissing the alarm when the cause has been fixed.

REFERENCES

Berge, J., Cottier, F., Last, K.S., Varpe, O., Leu, E., Soreide, J., Eiane, K., Falk-Petersen, S., Willis, K., Nygard, H., Vogedes, D., Griffiths, C., Johnsen, G., Lorentzen, D., and Brierley, A.S. (2009). Diel vertical migration of Arctic zooplankton during the polar night. *Biology Letters*, 5(1), 69–72.

Lønne, L. (2014). Development of a prototype sensor for measuring irradiance in the polar night.

Sakshaug, E., Johnsen, G.H., and Kovacs, K.M. (2009). *Ecosystem Barents Sea.* Tapir Academic Press.

A User manual

The following pages contain the user manual, meant for the end users of the system. It details installation of the server and the web application, and the user interface.

User manual for the SPC2 software package

Lars Lønne
`larslonn@stud.ntnu.no`

14 December, 2014

1 Installation

1.1 Connecting the camera to the server

1.1.1 Inputs

The SPC2 camera has five inputs. Only two of them will be used for this application. One is the power, which has no label, and the other is the USB. Figure 1 shows the inputs on the camera.

1.1.2 Connections

Connect the USB cable to the camera and to the server. Connect the power cable to the camera. The camera can take up to 10 seconds to start up after the power has been connected.

1.2 Databases

The server application needs at least one database to save measurements, settings and metadata in. The database must always be available, so failure preventions must be considered when implementing the database solution.

The chosen database system is MongoDB. It was chosen because it is an open-source, document driven database, which works particularly well with `node.js`, which is the platform that the server application is written in. It

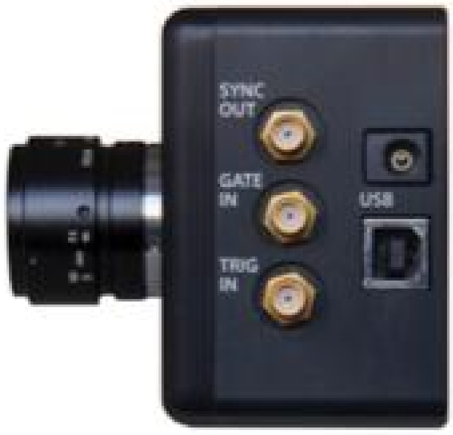


Figure 1: SPC2 inputs

also supports the notion of *replica sets*, which is an easy way of implementing redundancy.

This section describes how to set up redundant databases, and how to tell the server application which databases to use.

1.2.1 Getting MongoDB

MongoDB can be downloaded from the download page. Choose the correct download for your system and follow the installation instructions.

For many systems, such as Mac OS X and many flavors of Linux, MongoDB can also be installed through a package manager. See the MongoDB web page for details.

Once the installation is finished, you can proceed with the next section.

1.2.2 Replica sets

MongoDB supports redundancy through what it calls *replica sets*. This means that the application connects to two or more instances of the database at the same time. One of the databases is designated as the *primary*, and the others are *secondary*. The data is first saved to the primary database, and then replicated on the secondaries. In case of a failure, such as a power outage or a hardware failure, one of the secondaries is elected to be the new primary, and the system can continue in its normal mode of operation.

Important: Replica sets should always have an odd number of members, to ensure that elections proceed smoothly. Thus, 3 members should be sufficient to endure most problems.

The MongoDB manual recommends that replica set members in production environments maintain as much separation as possible. Storing the different databases on separate hard drives should be a minimum requirement. Running on separate machines is better, and on separate machines in separate locations would be ideal. Strive to separate the running databases as much as possible. **Never run all the instances from one hard drive**.

1.2.3 Starting `mongod`

A MongoDB database instance is started with the command `mongod`, which stands for *mongo daemon*. It takes a number of command line options, some of which are detailed here. The rest can be found on `mongod`'s help page, accessible with the command `mongod --help`. Options are specified on the command line, with two dashes preceding it, such as `mongod --option`.

`port <port-number>` specifies the port that the process will listen on. The default port is 27017, and should be used if the instances are running on separate machines. If the database processes are all running on one machine, they must listen on separate port numbers. Starting on 27017 and incrementing for each one is a good choice.

`dbpath <path>` specifies the path to the database files. This depends on the host machine, but `/data/db` is often used.

`replSet <set-name>` specifies the name of the replica set that the process belongs to. It can really be anything, as long as all the members belong to the same replica set. An example could be "spc2-repl-set".

`logpath <path>` saves the database logs to the file at `<path>` instead of printing it to the console. This should always be enabled. If you want the process to append to the logfile instead of overwriting it, use `--logappend`.

An example of how to start a `mongod` instance:

```
mongod --port 27018 --dbpath /data/db --replSet spc2-repl-set0
 --logpath /var/log/mongod.log
```

This would start a `mongod` instance, listening on port 27018, saving data at `/data/db`, belonging to the replica set `spc2-repl-set0` and saving its logs at `/var/log/mongod.log`.

1.2.4 Adding members to the replica set

Open a `mongo` shell on the machine that runs the primary process. This can be any of the processes. A `mongo` shell is opened by issuing the command

```
mongo --port <port-number>
```

`<port-number>` is the port number that the process on this machine listens on. E.g. if the process was started with

```
mongod --port 27018 ...
```

you would start the shell with the command

```
mongo --port 27018
```

Once the shell has started, run the command

```
rs.initiate()
```

This will initiate the replica set, and add this process as the primary. When it has finished, issue the following command for **each** of the other members in the replica set:

```
rs.add("<member-address>")
```

`<member-address>` is the address of the other process that was started. When all processes has been added to the replica set, run

```
rs.status()
```

to confirm that everything is set up correctly. All members in the set should be listed in the `members` property. If they are not, or you encounter any other errors, consult the documentation.

1.2.5 Creating the settings collection

For the server application to run, the `settings` and the `cameraspecs` collections must exist in the database. In the `scripts` directory, there is a script called `populate-db.js` that will create a settings object and save it to the database.

Open the script and edit the line that says

```
conn = new Mongo('spc2-server:27017')
```

Change `spc2-server` to the hostname of your machine, and change 27017 to the port your `mongod` process is running on.

Next, run the script with the following command:

```
mongo <hostname>:27017 scripts/populate-db.js
```

Change `<hostname>` and 27017 if your database is running on a different host and/or port. The `settings` and `cameraspecs` collections should now exist in the database.

1.2.6 The server application config file

When the databases are up and running, the addresses must be entered into the server application config file, so that the server can connect to them. Open the config file, and update it so it includes the following lines:

```
dbHostAddresses:
    - <address1:port1>
    - <address2:port2>
    - ...
databaseName: spc2db
```

Each member of the replica set must be listed on its own line under `dbHostAddresses`. Make sure that the file only includes one instance of the `dbHostAddresses` list and the `databaseName` variable.

1.3 Server

1.3.1 Prerequisites

The server is a node.js application with MongoDB databases. Setting up the databases is described in the database section. To run the server application, the node.js platform must be installed. See the node.js webpage for instructions.

Also, before proceding, the camera should be connected to the server.

1.3.2 The `config` file

The server's configuration file is called `config.yaml`, and should exist in the same directory as `server.js`. `server.js` is the main file of the server application.

The database specific entries in the configuration file are dealt with in the section on databases. Specifically, the variables `dbHostAddresses` and `databaseName` should exist and have its correct values.

The config file may include a variable called `port`, which specifies which port the server should listen for connections on. If `port` is not specified, the server defaults to listening on port 8080.

The config also has an entry named `sensorArea`, which should be `3.14e-10`. This is the surface area of the sensor in the camera, and is used by the server application for calculating the irradiance values.

1.3.3 Installing dependencies

The server application depends on a number of external modules to function. The dependencies are all listed in the file `package.json` and can be installed using `npm`; *the node package manager*. `npm` is installed with the node platform, and should be available on your system.

Run `npm install` to install all dependencies.

1.3.4 External dependencies

The server application depends on http://www.graphicsmagick.org for converting the test images from `TIFF` to `PNG` format. It must be installed on the machine that the server application is running on.

On an Ubuntu Linux server, it is installed like this:

```
sudo apt-get install graphicsmagick
```

If you are running a different system, consult the GraphicsMagick webpage.

1.3.5 Compiling the C API

The last step before starting the server is compiling the C API. This job is done with the tool `scons`. If it is not installed on your machine, install it now through your package manager.

`SConstruct` files are provided for both Mac OS X and Linux. Compile the C API by calling `scons` like this:

```
scons -f SConstruct.linux
```

Change `linux` to `osx` if you are running on a Mac.

1.3.6 Starting the server

The server application includes a startup script called `run-server.sh`. It is written in bash, and will run on all Mac OS X and Linux systems. It will not run on Windows, except through Cygwin or something similar.

The server is run using a program called `supervisor`, which restarts the server in case of failures. If this program is not installed, it can be installed with `npm`:

```
npm install -g supervisor
```

Then, run the script to start the server:

```
./run-server.sh
```

1.3.7 Running

The server should now be running. It will try to connect to the camera and the database, and log any errors to the console. Usual problems at this stage is not having the camera connected, or not having a running database process.

The server should now be up and running, and listening for HTTP requests on the configured port, or 8080 by default.

1.4 Web application

The web application is a couple of simple pages for controlling and monitoring the SPC2 camera, through the server. This section assumes that the server is running, and that you have the server address.

1.4.1 Setup

The web application requires minimal setup. First, it needs to know is the address to the server. This information must be entered in the file `js/config.js`, which is a standard javascript file.

Change the `host` and `port` to the address and port number of your server. Do not change the `/api` part. A sample `config.js` is shown below.

```
var host = 'http://129.241.143.245',
    port = 8080;

var config = {
  //
  apiUrl: host + ':' + port + '/api'
}
```

Second, the web application's dependencies must be installed. This is done in two steps, because it uses both `npm` and `bower` to handle its dependencies. Change to the top directory, where the file `package.json` is located. Run the command

```
npm install
```

This should install all the dependencies for node.js. Then, change to the `resources` directory and run the command

```
bower install
```

This should install all dependencies for the web application itself.

1.4.2 Hosting

The web application is also a node.js application, and can be started with the command

```
NODE_ENV=production node app.js
```

Important: It uses basic authentication to authenticate the user. If security is a concern, the web application must run on `https` only.

2 Web interface

The web interface is the main interface for end users. Through it, users can examine the latest measurements, set capture parameters and download data from the camera. This section describes each part of the web interface in detail, and explains how to use it.

2.1 The main page

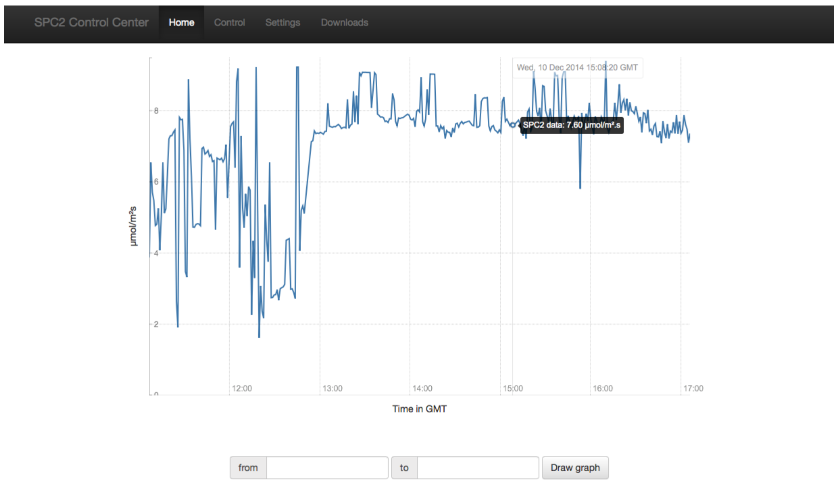


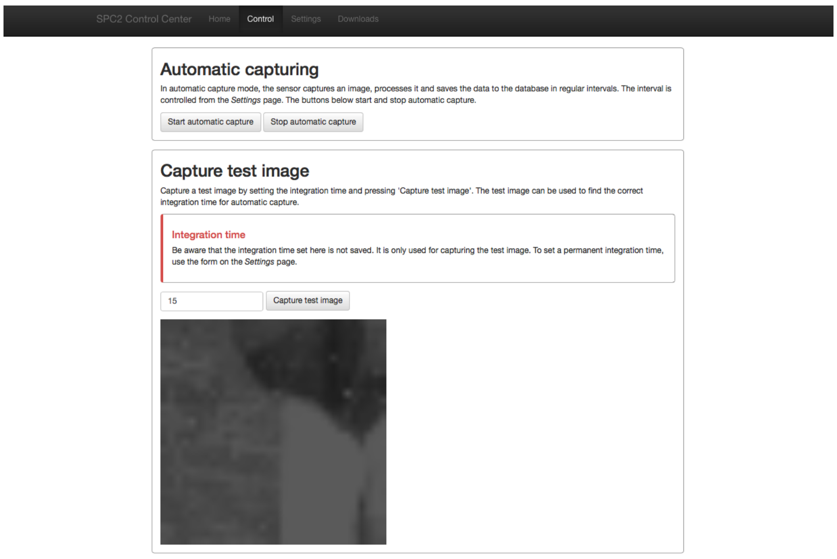
Figure 2: Main page

This is the main page of the application. By default, it shows a graph of the measurements taken in the past six hours. Time in GMT is on the x axis, while light measurements on the quantum scale is on the y axis.

By moving the mouse pointer over the graph, it is possible to read the exact measurement at that time instant.

Below the graph are the graph controls. By choosing the *from* and *to* date and clicking the *Draw graph* button, you can show measurements for a specific time range. Beware that all the data must be read from the database, so if you choose a large time range, it might take some time before the graph is shown again.

2.2 The Control page



Automatic capturing

In automatic capture mode, the sensor captures an image, processes it and saves the data to the database in regular intervals. The interval is controlled from the *Settings* page. The buttons below start and stop automatic capture.

Capture test image

Capture a test image by setting the integration time and pressing 'Capture test image'. The test image can be used to find the correct integration time for automatic capture.

Integration time

Be aware that the integration time set here is not saved. It is only used for capturing the test image. To set a permanent integration time, use the form on the *Settings* page.

15

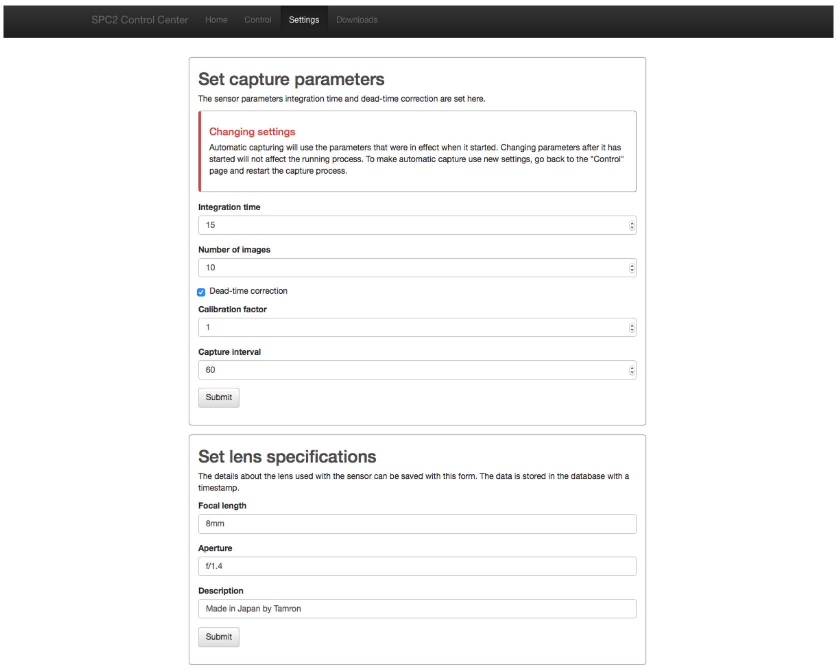
Figure 3: Control page

The *Control* page enables you to control the camera operations. There are two options, controlling *Automatic capturing* and capturing test images.

Automatic capturing is used for capturing time series of light measurements. It has two buttons, which starts and stops the automatic capturing mode. In this mode, the camera captures a series of images at given time intervals. The average of these images is stored in the database, along with the capture parameters that were active when the image was aquired. Also stored are the total photons per second that was detected, and the irradiance value on the quantum scale, micromol photons per square meter per second. The time interval is set on the *Settings* page.

Capture test image is for testing different integration times. Integration time is analogous to exposure on a traditional camera. Enter the integration time in milliseconds and press the button to capture an image. After the image has been captured and sent from the server, it will appear below the button.

2.3 The Settings page



Set capture parameters

The sensor parameters integration time and dead-time correction are set here.

Changing settings

Automatic capturing will use the parameters that were in effect when it started. Changing parameters after it has started will not affect the running process. To make automatic capture use new settings, go back to the "Control" page and restart the capture process.

Integration time: 15

Number of images: 10

[✓] Dead-time correction

Calibration factor: 1

Capture interval: 60

Set lens specifications

The details about the lens used with the sensor can be saved with this form. The data is stored in the database with a timestamp.

Focal length: 8mm

Aperture: f/1.4

Description: Made in Japan by Tamron

Figure 4: Settings page

The settings page allows you to set the capture parameters for the camera, and save information about the lens used with the camera.

2.3.1 Capture parameters

The SPC2 camera works is very different from a traditional camera, and this is reflected in the parameters. The camera actually counts the number of photons it detects during its exposure time. However, if the user is allowed to set the exposure time directly, the photon counters might overflow, and the results would be undefined. To prevent this, the camera has a constant exposure time of 20.74 microseconds. The data captured during this time is referred to as a *frame*.

To enable longer exposures than 20.74 microseconds, the camera sums multiple frames. This sum of frames is referred to as an image. Integration time indirectly specifies how many frames we want to sum in each image. The number of frames is set to the value that comes closest to the specified time. The integration time is in milliseconds.

Each measurement takes a number of images, and returns the average of those images. The number of images captured each time is specified in the input field *Number of images.*

Dead-time is the time from the camera detects a photon, until it is ready for detection again. *Dead-time correction* is an algorithm that tries to compensate for this unproductive time. It can be turned on and off with the checkbox.

Calibration factor is a factor that is multiplied with the calculated irradiance, before it is saved in the database. Use it if you need to calibrate the camera.

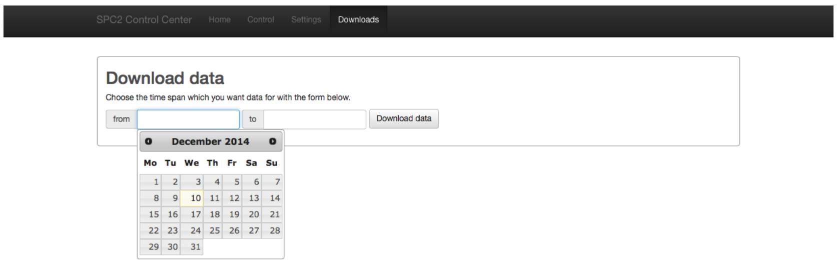
Capture interval is the amount of time between each measurement in automatic mode. The interval is in seconds.

Note: Although unlikely in practice, it is possible to set parameters such that each measurement takes longer than the interval between measurements. Avoid this.

2.3.2 Lens specifications

In this section you can record the specifics about the lens used with the camera. The information is stored in the database with a timestamp, such that it is possible to go back and see which lens was in use for the measurements.

2.4 The Download page



Download data

Choose the time span which you want data for with the form below.

Figure 5: Download page

This page contains only one form, that allows you to download measurements from the database. Fill in the *from* and *to* dates, and click *Download data.*

The data is a standard comma separated values file (CSV), with a header.

B Specification

The following pages contain the software specification. It is discussed in more detail in section 2.

Software specification

The purpose of this document is to specify a software system ("the program") for controlling the SPC2 Photon Imager ("the sensor").

Software functionality

The purpose of the sensor is to measure light by imaging the light field using photon counters. The raw output from the sensor is an image with 32×32 pixel resolution. Each of these pixels represent the photons counted by one counter during the exposure. I.e. the sensor consists of 1024 individual counters.

The main purpose of the program is to control the sensor. The program must be able to send commands to, and receive information from the sensor.

The following specifications are prioritized into three categories: *Priority A*, *Priority B* and *Priority C*. Functionality listed under *Priority A* is minimum functionality that must be implemented for the program to be functional. *Priority B* lists functions that are non-essential, but important for a good user experience. Functionality listed under *Priority C* are requests from the end users that most likely will not be implemented in this development cycle due to time constraints. *Priority C* functions will not be attempted until A and B are complete.

Capturing images

The sensor captures images of the light field in *frames*, which is defined as an exposure of 20.74 µs. The parameter *integration time* decides how many frames should be summed up in the output image from the sensor.

Dead-time is the time it takes to restore the photon detector after detection of a photon. During this time, no detection takes place. This effect introduces a non-linearity in the measurements, which can be reduced by a setting known as *dead-time correction.* This correction is most effective at low to moderate light intensities.

Priority A

- The program must enable the user to set and change *integration time* and *dead-time correction*.
- Focal length, field of view and aperture all depend on the lens used and are not controlled by the program. The program must include a text field where info about the lens can be entered. This info will be saved in the database with the captured data.
- Shutter speed is controlled indirectly by the *integration time*.
- The program must set default values for all parameters, and the program should be able to reset the parameters to the default values. Examples of default values could be *integration time* $= 0.5\,\mathrm{s}$, and *dead-time correction* on. Actual values will be decided after testing the sensor in its operational environment.
- The program must store the values for all parameters, i.e. the parameters must not change or be reset by any other means than user interaction.
- The program must be able to capture images in automatic mode. Automatic mode means that the program captures an image every t seconds, where t is specified by the user.

Priority B

- Enable the user to capture test images. A test image is an image captured at the command of the user. The program may then show the image to the user, along with data connected to the image, such as the irradiance in quanta or energy mode.

Storing data

Priority A

- The program must store all data from the sensor in a reliable and secure manner. Each captured image, or equivalent data, must be stored.
- The stored data should be calibrated to the quantum scale, i.e. it should be stored as $\mu\mathrm{mol/m^2s}$.
- The parameters of the sensor must be stored with the data, for future recalculation and confirmation.
- A calibration factor, if calculated, should be stored with the captured data. This way, the original measurements (raw counts) are always available.

- The storing of data must be robust, meaning that captured data from the sensor must be stored, even if there exists some problem with the database or the program.

Priority B

- Implement alarms to warn the user about storage problems.
- Also store data calibrated to the energy scale, i.e. µW/m^2.

Access to stored data

Priority A

- The program must enable the user to access data stored in the database. The program must offer the user to download the entire database, either as a database file, or as a dump of the database.
- The program should display the captured data through a web interface in the form of a plot. The units on the displayed data should be µmol/m^{2}s.

Priority B

- The program should also make parts of the collected data available, such as data captured within a specified timeframe. This timeframe is specified by the user. The data can then be downloaded as a structured plain text file, such as CSV.

Remote access

Priority A

- The program must enable the user remote access to the sensor. This means that the user must be able to access the captured data over a network, either the internet or a local network.
- Read access must be provided remotely, i.e. the user must be able to inspect and download captured data.

Priority B

- Controlling the sensor, e.g. by setting parameters, may be provided remotely.

Security

Priority A

- The program must be secured against unauthorized access.
- The program must ensure that the user is authenticated before the user is allowed to download data or set sensor parameters.

Priority B

- Access to view the captured data may be allowed without authentication.

Hardware functionality

Power supply

Priority B

- Review the available power supply.
- Plan a robust and failsafe system for continuous power supply to the sensor.

Priority C

- Implement the planned power supply system.

Future adaptation: Underwater operations

Priority C

- Review systems and methods for adapting the sensor to underwater operations.
- Plan/design a system for adapting the sensor to underwater operations, e.g. on landers and mobile platforms such as AUV/ROV.

C Code documentation

This appendix includes the documented code for the server application. The documentation was autogenerated with *docco*[12], an open source documentation generator.

The code documentation is primarily intended for future developers and maintainers, who want to understand and extend the code.

The code is also included in the accompanying zip archive.

[12]http://jashkenas.github.io/docco/

server.js

server.js

```
var express = require('express');
var app = express();
var basicAuth = require('basic-auth');
var async = require('async');
var bodyParser = require('body-parser');
var confert = require('confert')
var config = confert('config');
var winston = require('winston');
var apiRouter = express.Router();
var jsonParser = bodyParser.json();
var fs = require('fs');
```

`sdk`, `dataAccess` and `processor` are local modules that handle the connection to the camera, access to the database and intermediate processing of data, respectively.

```
var sdk = require('./modules/spc2-sdk/spc2-sdk.js');
var dataAccess = require('./modules/data-access/data-access.js');
var processor = require('./modules/data-processor/data-processor.js');
var autoCapture;

var logger = new (winston.Logger)({
  transports: [
    new (winston.transports.Console)({level: 'debug', colorize: true})
  ]
});
```

HTTP access control (CORS)

The following middleware adds the *Access-Control-Allow-Origin* to every response. The purpose of this is to allow clients to make requests to this domain, which is different from the domain the request was made from. For more information, see this article on MDN.

```
app.use(function(req, res, next) {
  res.header("Access-Control-Allow-Origin", "*");
  res.header("Access-Control-Allow-Headers",
             "Origin, X-Requested-With, Content-Type, Accept");
  next();
});
```

Application Programming Interface

The rest of the file sets up the routes for the API, and defines the functions to be called for requests to those routes. Common for all of the routes is that they respond with `200` on success, and either `400` or `500` for errors on the client or server side, respectively.

Use the `apiRouter` for all calls to addresses starting with `/api`.

```
app.use('/api', apiRouter);
```

Checks for a connection to the camera. Responds with `200` if successful, and `500` if a connection does not exist, and a new one can not be made.

```
apiRouter.get('/connected', function(req, res) {
  var connected = sdk.init();
  res.sendStatus(connected ? 200 : 500);
});
```

Disconnects from the camera. Note that this route responds to `POST` requests, while `/connected` responds to `GET`. The reason for this is reflected in the name of the routes, also. `/connected` is meant as a status request, as in the question *is the camera connected?* `/disconnect` on the other hand, assumes that the camera is connected, and that the user issues the command *disconnect the camera.*

```
apiRouter.post('/disconnect', function(req, res) {
  var disconnected = sdk.destroy();
  res.sendStatus(disconnected ? 200 : 500);
});
```

Gets data from the database, between the dates given in the route name. `from` and `to` must be on a form that is recognized by `Date.parse`. The recommended format is the ISO 8601 format, because it is clear and unambiguous.

On success, the data is written to a CSV file and offered to the user as a download.

```
apiRouter.get('/getData/:from/:to', function(req, res) {
  var from = new Date(req.params.from);
  var to = new Date(req.params.to);
  if (!from || !to) {
    var msg = 'Date format passed to /getData is not valid.';
    logger.warn(msg);
    res.status(400).send(msg);
  } else {
    var data = dataAccess.getData(from, to, processData);
  }

  function processData(err, results) {
    if (err)
      return logger.error(err);

    processor.createCSV(results, downloadFile);
  }

  function downloadFile(err, path) {
    if (err)
      return logger.error(err);

    var filename = 'data.csv';
    res.download(path, filename, function(err) {
      processor.deleteFile(path);
    });
  }
});
```

Calls to this route returns the same data as calls to `/getData`, but in JSON format. It is used for plotting data in the web application.

```
apiRouter.get('/plotData/:from/:to', function(req, res) {
  var from = new Date(req.params.from);
  var to = new Date(req.params.to);
  if (!from || !to) {
    var msg = 'Date format passed to /plotData is not valid.';
    logger.warn(msg);
    res.status(400).send(msg);
  } else {
    var data = dataAccess.getData(from, to, function(err, results) {
      if (err) return logger.error(err);

      var plotData = [];
      results.forEach(function(result) {
        var time = result.time.getTime(),
            value = result.quantumScale;
        plotData.push({ time: time, value: value });
      });
      logger.info('Sending plotdata');
      res.send(plotData);
    });
  }
});
```

Sets parameters for the camera, and stores the parameters in the database. This route expects the parameters as a JSON object included in the request. The object should have the properties `integrationTime` and `nImages`. `integrationTime` is the exposure time of the camera in milliseconds, and `nImages` is the number of images captured in each snap. The actual measurement stored in the database is the average of these `nImages`.

```
apiRouter.post('/setParameters', jsonParser, function(req, res) {
  var args = req.body;
  if (!args) {
    return res.sendStatus(400);
  }

  var integrationTime = parseFloat(args.integrationTime, 10);
  var nImages = parseInt(args.nImages, 10);

  if (!integrationTime || !nImages) {
    var errMsg = 'Params to /setParameters are not valid numbers.';
    logger.error(errMsg);
    return res.status(400).send({error: errMsg});
  }
```

The camera operates in *normal mode*, where one exposure is 20.74 microseconds.

```
  var exposureTime = 20.74e-6;
  var nIntegratedFrames = Math.round(integrationTime/exposureTime);

  async.parallel([
    function setCameraParameters(callback) {
      var status = sdk.setCameraParameters(nIntegratedFrames, nImages);
      callback(null, status);
    },
    function saveParameters(callback) {
      var settings = {};
      settings.numIntegratedFrames = nIntegratedFrames;
      settings.numImages = nImages;
      dataAccess.updateSettings(settings, callback);
    }
  ], function after(err, results) {
    if (err) return logger.error(err);
    var status = results[0];
    res.status(status.success ? 200 : 500).send(status);
  });
});
```

Sets the interval between each measurement when in automatic capture mode. The `POST` request must include a JSON object with the property `captureInterval`. The interval is given in seconds.

```
apiRouter.post('/setCaptureInterval', jsonParser, function(req, res) {
  var postData = req.body;
  if (!postData || !postData.captureInterval) {
    logger.error('Invalid data sent to /setCaptureInterval');
    logger.debug('postData: ' + postData);
    return res.sendStatus(400);
  }
  dataAccess.updateSettings({
    secondsBetweenCaptures: postData.captureInterval
  }, function callback(err) {
    if (err) {
      logger.error(err);
      res.sendStatus(500);
    }
    logger.info('Updated capture interval');
    res.sendStatus(200);
  });
});
```

Sets the calibration factor for the measurements. Each measurement is multiplied with the calibration factor before it is stored in the database. The calibration factor is stored with each measurement.

```
apiRouter.post('/setCalibrationFactor', jsonParser, function(req, res) {
  var postData = req.body;
  if (!postData || !postData.calibrationFactor) {
    logger.error('Invalid data sent to /setCalibrationFactor');
    logger.debug('postData: ' + postData);
    return res.sendStatus(400);
  }
  dataAccess.updateSettings({
    calibrationFactor: postData.calibrationFactor
  }, function callback(err) {
    if (err) {
      logger.error(err);
      res.sendStatus(500);
    }
    logger.info('Set calibration factor');
    res.sendStatus(200);
  });
});
```

Enables the dead-time correction algorithm. After a sensor has detected a photon, it needs to reset before it is ready for detection again. During this reset, no detections will be made. This time is called the *dead-time*. Dead-time correction is an algorithm that tries to correct the errors in the measurements due to the dead-time.

```
apiRouter.post('/enableDeadTimeCorrection', function(req, res) {
  var statusObj = sdk.setDeadTimeCorrection(true);
  dataAccess.updateSettings({
    deadTimeCorrection: true
  }, function(err) {
    if (err) {
      logger.error('Could not update dead-time correction in settings: ' + err);
      res.sendStatus(500);
    }
    res.status(statusObj.success ? 200 : 500).send(statusObj);
  });
});
```

Disables the dead-time correction.

```
apiRouter.post('/disableDeadTimeCorrection', function(req, res) {
  var statusObj = sdk.setDeadTimeCorrection(false);
  dataAccess.updateSettings({
    deadTimeCorrection: false
  }, function(err) {
    if (err) {
      logger.error('Could not update dead-time correction in settings: ' + err);
      res.sendStatus(500);
    }
    res.status(statusObj.success ? 200 : 500).send(statusObj);
  });
});
```

Captures a single image and returns a JSON object, containing the image data as a base64 encoded string. The image data is in the `imageBase64` property.

```
apiRouter.get('/getTestImage', function(req, res) {
```

Expects a query parameter `intTime`, which specifies the integration time in milliseconds.

```
  var intTime = req.query.intTime * 1;
  if (!intTime || intTime === NaN || intTime <= 0) {
    logger.error('/getTestImage: invalid parameter intTime: ' + intTime);
    res.sendStatus(400);
  }
  var nIntFrames = time2Frames(intTime);
  sdk.captureTestImage(nIntFrames, function(path) {
    fs.readFile(path, function(err, data) {
      if (err) {
        logger.error('/getTestImage: Could not read image file: ' + err);
        return res.sendStatus(500);
      }
      var base64Data = data.toString('base64');
      res.status(200).send({
        imageBase64: base64Data
      });
    });
  });
});
```

Starts automatic capturing. Automatic capturing means that the camera captures a set of images at set intervals. The average of these images are stored in the database as a measurement, along with the capture parameters. Most of the functionality of this route is encapsulated in the `autoCapture` closure.

```
apiRouter.post('/startAutoCapture', function(req, res) {
  logger.info('Starting autocapture');
  autoCapture.start();
  res.sendStatus(200);
});
```

Stops automatic capturing.

```
apiRouter.post('/stopAutoCapture', function(req, res) {
  logger.info('Stopping autocapture');
  autoCapture.stop();
  res.sendStatus(200);
});
```

The user can store metadata about the camera in the database for later reference. This route accepts the meta data as a JSON object, and saves it to the database. The JSON object may include the properties `focalLength`, `aperture` and `description`.

```
apiRouter.post('/saveCameraSpecs', jsonParser, function(req, res) {
  var postData = req.body;
  if (!postData) {
    logger.error('Invalid data or no data posted to /saveCameraSpecs');
    return res.sendStatus(400);
  }
  dataAccess.saveCameraSpecs(postData, function(err) {
    if (err) {
      logger.error('Could not save camera specs: ' + err);
      res.sendStatus(500);
    } else {
      logger.info('New camera specs saved');
      res.sendStatus(200);
    }
  });
});
```

Returns the stored settings from the database. Settings include the number of integrated frames, the number of images captured in each snap, seconds between each capture in automatic mode, the calibration factor and a boolean telling us if dead-time correction is enabled or not.

```
apiRouter.get('/getSettings', function(req, res) {
  dataAccess.getSettings(function gotSettings(err, result) {
    if (err) {
      logger.error('Could not get settings: ' + err);
      res.sendStatus(500);
    } else {
      logger.debug('Getting settings object');
      res.status(200).send(result);
    }
  });
});
```

Returns the stored camera specifications from the database. These are the specifications stored with `/saveCameraSpecs`.

```
apiRouter.get('/getCameraSpecs', function(req, res) {
  dataAccess.getCameraSpecs(function gotSpecs(err, result) {
    if (err) {
      logger.error('Could not get camera specs: ', err);
      res.sendStatus(500);
    } else {
      logger.debug('Getting camera specs');
      res.status(200).send(result);
    }
  });
});
```

Startup

Creates the server instance and starts listening for requests.

```
var listenOn = config.port || 8080;
var server = app.listen(listenOn, function () {
```

Attempt to connect to the camera.

```
  var connected = sdk.init();
  if (!connected) {
    logger.error('Could not connect to camera. Exiting.');
    process.exit(1);
  }
```

The following functions are called in series, because the database must be connected before the other functions can be called.

```
  async.series([
    function connectToDatabase(callback) {
      dataAccess.connect(config.dbHostAddresses, config.databaseName, callback);
    },
```

This creates the autoCapture closure. The reason for using a closure, is to encapsulate the `intervalObject`, which is needed for stopping a running timer.

```
    function createAutoCaptureClosure(callback) {
      autoCapture = (function() {
        var intervalObject;
        return {
          start: function() {
            dataAccess.getSettings(function(err, settings) {
              if (err)
                return logger.error('Could not start auto capture: ' + err);
```

The timer function uses the sdk to capture an image and saves the data to the database. The camera parameters are saved along with the measurement.

```
          intervalObject = setInterval(function() {
            var measurement = sdk.captureImage();
            measurement.numIntegratedFrames = settings.numIntegratedFrames;
            measurement.numImages = settings.numImages;
            measurement.photonsPerSecond =
              processor.getPhotonsPerSecond(settings.numIntegratedFrames,
                                            measurement.data);
            measurement.quantumScale =
              processor.convertToQuantumScale(measurement.photonsPerSecond,
                                              settings.calibrationFactor);
            measurement.calibrationFactor = settings.calibrationFactor;
            dataAccess.saveMeasurement(
              measurement,
              function(err, m) {
                if (err) return logger.error(err);
                else return logger.debug('Measurement saved at ' + m.time);
              });
          }, settings.secondsBetweenCaptures * 1000);
        })
      },
      stop: function() {
        clearInterval(intervalObject);
      }
    };
  })();
  callback(null);
},
function setStoredParameters(callback) {
```

Gets settings from database and apply them to camera. If any errors occurs here, the application assumes that either the camera is not connected, or the database is not running. Either way, there is no point for the application to start up, so it just exits.

```
  dataAccess.getSettings(gotSettings);

  function gotSettings(err, settings) {
    if (err) {
      logger.error('Could not get settings from database: ' + err);
      process.exit(1);
    }
```

```
        var status = sdk.setCameraParameters(settings.numIntegratedFrames,
                                             settings.numImages);
        if (!status.success) {
          logger.error('Could not set camera parameters');
          process.exit(1);
        }

        status = sdk.setDeadTimeCorrection(settings.deadTimeCorrection);
        if (!status.success) {
          logger.error('Could not set dead-time correction');
          process.exit(1);
        }

        logger.info('Camera parameters have been set');
        callback(null);
      }
    }], function initDone(err) {
      logger.info('Initialization done.');
    });

  var host = server.address().address;
  var port = server.address().port;
  logger.info('Server listening at http://%s:%s', host, port);
});
```

Returns the number of integrated frames that comes closest to the given integration time. *integrationTime* is the time in milliseconds.

```
function time2Frames(integrationTime) {
  var micro = 1e-6,
      milli = 1e-3;
  var exposureTime = 20.74 * micro;
  var timeInSeconds = integrationTime * milli;
  var frames = Math.round(timeInSeconds / exposureTime);
  return frames;
}
```

data-access.js

data-access.js

Purpose

This module includes functions that read and write to the database. It acts as an abstraction layer between the server application and the database, hiding the database details from view.

MongoDB

The server uses MongoDB as its database. Mongoose is a node.js module for interfacing with MongoDB.

```
var mongoose = require('mongoose');
var db;

var winston = require('winston');
var logger = new (winston.Logger)({
  transports: [
    new (winston.transports.Console)({level: 'debug', colorize: true})
  ]
});
```

Module functions

```
var dataAccess = module.exports;
```

Connects to the database(s). Addresses is a list of addresses to running mongod instances. The use of multiple databases assures redundancy in case some of them fail.

```
function connect(addresses, databaseName, callback) {
```

Concatenates the addresses to one URI that mongoose accepts.

```
  var uri = concatAddresses(addresses, databaseName);
  var options = {
```

Sets `keepalive` to `true`, to make sure that the database connection doesn't time out.

```
    server: {
      socketOptions: {
        keepAlive: 1
      }
    },
    replset: {
      socketOptions: {
        keepAlive: 1
      }
    }
  };
  mongoose.connect(uri, options);
  db = mongoose.connection;
  db.on('error', logger.error);
```

Asynchronously sets up the database schemas, once the connection has been made.

```
  db.once('open', function defineModels() {
```

The schema for a measurement. `data` holds the original data from the camera, and `time` holds the timestamp from when the measurement was made. The rest of the fields are saved with the measurement to make sure the results are reproducible.

```
    var Measurement = mongoose.Schema({
      data: [{type: Number, min: 0}],
      time: Date,
      numIntegratedFrames: Number,
      numImages: Number,
      photonsPerSecond: Number,
      quantumScale: Number,
      calibrationFactor: Number
    });
    mongoose.model('Measurement', Measurement);
```

Defines the settings schema. Settings include the camera parameters, the capture interval and the calibration factor.

```
    var Settings = mongoose.Schema({
      numIntegratedFrames: { type: Number, min: 1 },
      numImages: { type: Number, min: 1 },
      secondsBetweenCaptures: Number,
      calibrationFactor: { type: Number, min: 0 },
      deadTimeCorrection: Boolean
    });
    mongoose.model('Settings', Settings);
```

Schema for saving metadata about the camera.

```
    var CameraSpecs = mongoose.Schema({
      time: Date,
      focalLength: String,
      aperture: String,
      description: String
    });
    mongoose.model('CameraSpecs', CameraSpecs);

    if (callback && typeof callback === 'function')
      callback(null);
  });
};
```

Disconnects from the database.

```
function disconnect() {
  mongoose.connection.close();
}
```

Concatenates multiple addresses into one string. Used for connecting with mongoose.

```
function concatAddresses(addresses, databaseName) {
  var pattern = /^(mongodb:\/\/)?(\S+)/;
  var arr = addresses.reduce(function(acc, address) {
    var matches = pattern.exec(address);
    return acc.concat(matches[2]);
  }, []);
  return 'mongodb://' + arr.join(',') + '/' + databaseName;
}
```

Standard get functions.

```
function getMeasurementModel() {
  return mongoose.models.Measurement;
}

function getSettingsModel() {
  return mongoose.models.Settings;
}

function getCameraSpecsModel() {
  return mongoose.models.CameraSpecs;
}
```

Saves a new measurement to the database. Saves are done asynchronously, and the provided callback is called when the save is complete.

```
function saveMeasurement(measurement, callback) {
  var Measurement = getMeasurementModel();
  var m = new Measurement(measurement);
  m.save(callback);
}
```

Similar to `saveMeasurement`, but updates instead of saving a new object.

```
function updateSettings(newSettings, callback) {
  var Settings = getSettingsModel();
  Settings.update({},
                  { $set: newSettings },
```

`upsert` means to insert if a record does not exist, and update otherwise.

```
                  { upsert: true },
                  callback);
}
```

Identical to `saveMeasurement`, but for camera specs.

```
function saveCameraSpecs(specs, callback) {
  var CameraSpecs = getCameraSpecsModel();
  var cs = new CameraSpecs(specs);
  cs.save(callback);
}
```

Reads the settings from the database.

```
function getSettings(callback) {
  logger.debug('called getSettings');
  getSettingsModel().findOne({}, callback);
}
```

Reads the camera specs from the database. Each set of specs are saved by itself, with a timestamp. This function gets the last one.

```
function getCameraSpecs(callback) {
  getCameraSpecsModel()
    .find()
    .sort('-time')
    .limit(1)
    .exec(callback);
}
```

Get measurements taken between `from` and `to`. All parameters must be specified.

```
function getData(from, to, callback) {
  if (!from || !to || !callback) {
    return logger.error('data-access: getData called with invalid arguments.');
  }

  logger.info('Getting measurements between ' + from.toUTCString() +
              ' and ' + to.toUTCString());
  var model = getMeasurementModel();
  model.find({ time: { $gte: from, $lt: to } }, callback);
}
```

Defines the interface to this module. `dataAccess` is an alias for `module.exports`, which is the object exported from a node.js module.

```
dataAccess.connect = connect;
dataAccess.disconnect = disconnect;
dataAccess.getMeasurementModel = getMeasurementModel;
dataAccess.getSettingsModel = getSettingsModel;
dataAccess.getCameraSpecsModel = getCameraSpecsModel;
dataAccess.saveMeasurement = saveMeasurement;
dataAccess.updateSettings = updateSettings;
dataAccess.getSettings = getSettings;
dataAccess.getCameraSpecs = getCameraSpecs;
dataAccess.getData = getData;
dataAccess.saveCameraSpecs = saveCameraSpecs;
```

data-processor.js

data-processor.js

Purpose

This module is a collection of functions that manipulate data in some way, and where the function are too large to include in the other modules without violating the single responsibility principle. Conversions and calculations on the data typically belongs here.

```
var winston = require('winston');
var logger = new (winston.Logger)({
  transports: [
    new (winston.transports.Console)({level: 'debug', colorize: true})
  ]
});
var config = require('confert')(process.cwd() + '/config');
var tmp = require('tmp'),
    fs = require('fs'),
    async = require('async'),
    os = require('os');

var processor = module.exports;

function sum(coll) {
  return coll.reduce(add);
}

function add(x, y) {
  return x + y;
}
```

Module functions

Calculates the number of photons per second measured in an image. The `image` is an array, where each element represents one photon detector in the camera.

```
processor.getPhotonsPerSecond =
  function photonsPerSecond(numIntegFrames, image) {
    if (numIntegFrames <= 0) {
      logger.error('Called photonsPerSecond with invalid numIntegFrames');
      return 0;
    }
```

In *normal* mode, the exposure time is always a multiple of 20.74 microseconds.

```
    var exposureTime = numIntegFrames * 20.74e-6;
    var detectedPhotons = sum(image);
    return detectedPhotons/exposureTime;
  };
```

Converts the photons per second count to the quantum scale (micromol photons per square meter per second). Requires that we know the sensor area, which is specified in the config file.

```
processor.convertToQuantumScale =
  function quantumScale(photonsPerSecond, calibrationFactor) {
    var avogadro = 6.022141e+23;

    var area = config.sensorArea || 0;
    if (area === 0) {
      logger.warning('Sensor area is not defined in the config file.' +
                     'Returning 0 for the quantum scale.');
      return 0;
    }

    return photonsPerSecond * 1e6/avogadro / area * calibrationFactor;
  };
```

Builds a standard CSV (comma separated values) file, with `,` as the separator, and a descriptive header. `results` is an array of measurement objects from the database. **Note**: Measurements are not sorted.

```
processor.createCSV =
  function createCSV(results, callback) {
    var header = 'time,numIntegratedFrames,numImages,photonsPerSecond,'
```

```
        + 'quantumScale,calibrationFactor' + os.EOL;
    tmp.file(function tmpFileCreated(err, path) {
      if (err)
        return logger.err(err);

      logger.debug('Created temp file ' + path);

      var stream = fs.createWriteStream(path, {
        flags: 'w',
        encoding: 'utf8'
      });

      stream.write(header);
```

Process each measurement object in the results. There is no guarantee that the results will be processed in order, ref. the async module.

```
      async.each(results, addResultToFile, function(err) {
        if (err)
          logger.error('createCSV: Could not write to stream: ' + err);
        stream.end();
      });

      stream.on('finish', function() {
        callback(null, path);
      });
```

Called for each measurement object in the `results` array. Assembles a CSV line and writes it to the file.

```
      function addResultToFile(result, cb) {
        delete result.data;
        var s = '' +
            result.time.toISOString() + ',' +
            result.numIntegratedFrames + ',' +
            result.numImages + ',' +
            result.photonsPerSecond + ',' +
            result.quantumScale + ',' +
            result.calibrationFactor + os.EOL;
        stream.write(s);
        cb();
      }
    });
  };
```

Deletes/unlinks a file. Typically used to delete a file created with the `createCSV` function, when the user has downloaded it.

```
processor.deleteFile =
  function deleteFile(path) {
    fs.unlink(path, function(err) {
      if (err)
        return logger.error('Could not unlink file: ' + err);

      logger.debug('Unlinked file: ' + path);
    });
  };
```

spc2-sdk.js

spc2-sdk.js

Purpose

This is the translation of the SDK from C to javascript. This module uses the module `ffi`, which stands for *foreign function interface*. It lets us create javascript functions from the C functions, by defining them as parameters to the `ffi.Library` function.

External modules

```
var ffi = require('ffi');
```

`ref` and `ref-array` simplifies handling of C pointers and arrays from javascript, by using its predefined `types`.

```
var ref = require('ref');
var ArrayType = require('ref-array');
var winston = require('winston');
var logger = new (winston.Logger)({
  transports: [
    new (winston.transports.Console)({level: 'debug', colorize: true})
  ]
});
```

`tmp` is a module for creating temporary files.

```
var tmp = require('tmp');
```

`gm` is a module for working with images. It requires GraphicsMagick to be installed.

```
var gm = require('gm');
```

Type definitions

Here we define the C types and C array types that we're going to need in the library function calls.

```
var ushort = ref.types.ushort;
var double = ref.types.double;
var DoubleArray = ArrayType(double);
var UShortArray = ArrayType(ushort);
```

Creating the javascript functions

The call to `ffi.Library` creates a javascript object, which contains javascript functions for all the C library functions that were specified in the parameters. After this call, the functions can be called just like any other javascript function. Each library function is defined with its name, return type and the types of its arguments.

```
var lib = ffi.Library('./libapi', {
  'init': ['bool', []],
  'destroy': ['bool', []],
  'apply_settings': ['bool', []],
  'set_background_img': ['bool', [UShortArray]],
  'set_background_subtraction': ['bool', ['bool']],
  'set_camera_par': ['bool', ['ushort', 'uint32']],
  'set_deadtime': ['bool', ['ushort']],
  'set_deadtime_correction': ['bool', ['bool']],
  'set_live_mode': ['bool', ['bool']],
  'get_img_position': ['bool', [UShortArray, 'uint32']],
  'get_live_image': ['bool', [UShortArray]],
  'get_snap': ['bool', []],
  'prepare_snap': ['bool', []],
  'average_image': ['bool', [DoubleArray]],
  'save_image_to_disk': ['bool', ['uint32', 'uint32', 'string', 'string']]
});
```

Module functions

```
var sdk = module.exports;
```

Connect to the camera.

```
sdk.init = function init() {
  logger.info('Connecting/checking connection to camera');
  return lib.init();
};
```

Disconnect from the camera.

```
sdk.destroy = function destroy() {
  return lib.destroy();
};
```

Enables or disables dead-time correction. The argument `enable` is a boolean which decides wether to enable (`true`) or disable (`false`) the correction algorithm.

```
sdk.setDeadTimeCorrection = function setDeadTimeCorrection(enable) {
  var success = true;
  success = success && lib.set_deadtime_correction(enable);
  success = success && lib.apply_settings();
  var statusObj = { success: success };
  if (!success) {
    var verb = enable ? 'enable' : 'disable';
    var errMsg = 'Could not ' + verb + ' dead-time correction.';
    logger.error(errMsg);
    statusObj.error = errMsg;
  }
  return statusObj;
};
```

Sets the camera's capture parameters; the number of integrated frames, and the number of images per snap.

```
sdk.setCameraParameters = function setCameraParameters(nIntegratedFrames, nImages) {
  var success = lib.set_camera_par(nIntegratedFrames,
                           nImages);
  if (!success) {
    logger.error('Could not set camera parameters.');
  }
  success = lib.apply_settings();
  if (!success) {
    logger.error('Could not apply camera settings.');
  }
  return {success: success};
};
```

Captures an image according to the parameters set with `sdk.setCameraParameters`. The returned image is an array of values, each value representing one photon detector in the camera. The values are the averages of all images captured in the snap.

```
sdk.captureImage = function captureImage() {
  var time = new Date();

  prepareAndGetSnap();

  var image = new DoubleArray(1024);
  var gotImage = lib.average_image(image);
  if (!gotImage) return logger.error('Could not get average image of snap.');

  var arr = new Array(1024);
  for (var i = 0; i < 1024; i++) {
    arr[i] = image[i];
  }

  return {
    data: arr,
    time: time
  };
};
```

This function also captures an image, like `sdk.captureImage`, but its purpose is a little different. Instead of capturing multiple images and returning the average data, this function captures just one image, and returns the image data as a `tiff` file. It also accepts parameters as input. It is used for capturing test images, to test the parameters before enabling automatic capture mode.

```
sdk.captureTestImage =
  function captureTestImage(nIntegratedFrames, callback) {
    if (!nIntegratedFrames) {
      logger.error('captureTestImage missing parameters');
      return {};
    }

    sdk.setCameraParameters(nIntegratedFrames, 1);
    prepareAndGetSnap();

    tmp.file(function fileCreated(err, path) {
      if (err) return logger.error('captureTestImage: Could not create tmp file.');

      var startImg = 1,
          endImg = 1,
          fileFormat = 'tiff-no-compression';
      var imageSaved = lib.save_image_to_disk(startImg, endImg, path, fileFormat);

      if (!imageSaved) {
        logger.error('captureTestImage: Could not save test image to disk.');
        callback('');
      } else {
        logger.info('captureTestImage: Test image saved.');
        var pngPath = path + '.png';
        gm(path).write(pngPath, function imageConverted(err) {
          if (err) {
            logger.error('Could not convert image to png: ' + err);
            return callback('');
          }
          logger.info('captureTestImage: Image converted');
          callback(pngPath);
        });
      }
    });
  }
```

Helper function. Used in both `sdk.captureImage` and `sdk.captureTestImage`.

```
function prepareAndGetSnap() {
  var prepared = lib.prepare_snap();
  if (!prepared) return logger.error('Could not prepare camera for snap.');

  var gotSnap = lib.get_snap();
  if (!gotSnap) return logger.error('Could not get snap.');
}
```